JPA Eficaz

As melhores práticas de persistência de dados em Java

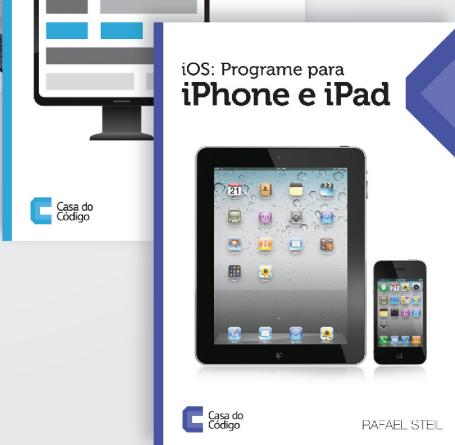
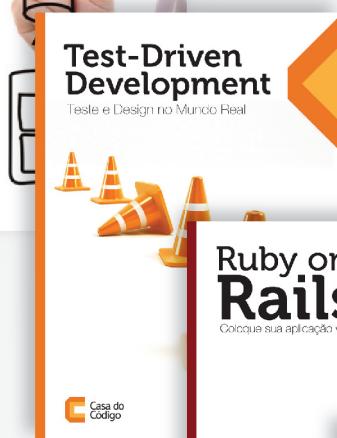
HÉBERT COELHO

Uma editora de livros técnicos feita por desenvolvedores para desenvolvedores.

Inscreva-se em nossa newsletter e receba novidades e lançamentos

www.casadocodigo.com.br/newsletter

Curta nossa fanpage no Facebook

www.facebook.com/casadocodigo

Caelum:
Cursos de TI presenciais e online

www.caelum.com.br

Dê seu feedback sobre o livro. Escreva para contato@casadocodigo.com.br

Sobre o autor

Hébert Coelho de Oliveira é analista desenvolvedor sênior, pós-graduado em Engenharia de Software, professor em faculdade e escritor nas horas vagas. Trabalha há mais de 10 anos com desenvolvimento de softwares e possui as certificações SCJP, SCWCD, OCBCD, OCJPAD.

É autor do livro JSF Eficaz, publicado pela editora Casa do Código, que dá dicas e melhores práticas para os desenvolvedores que utilizam o JSF em seus projetos.

É criador do blog http://uaiHebert.com, visualizado por 180 países, totalizando mais de 800 mil visualizações em seus 3 anos de vida. É ainda autor do framework EasyCriteria (http://easycriteria.uaihebert.com) que ajuda na utilização da Criteria da JPA, sendo testado com Hibernate, OpenJPA e EclipseLink e com 100% de cobertura nos testes.

Foi revisor de um livro específico sobre Primefaces e criador de posts em seu blog com aplicações completas utilizando JSF. Escreveu dois posts sobre JPA com diversas dicas que já passaram de 55 mil visualizações, e que também foi o ponto de partida desse livro.

Pós-graduado em MIT Engenharia de Software — desenvolvimento em Java pela Infnet RJ. Atualmente atua como professor para o curso de pós-graduação, ensinando o conteúdo de Java Web (JSP, Servlet, JSF e Struts) e tópicos avançados, como EJB, Spring e WebServices.

Agradecimentos

Agradeço a Deus pela sabedoria, força de vontade e inteligência para conseguir finalizar o livro.

Dedico esse livro àquela que é o maior presente que Deus poderia me dar, minha esposa Daiane. Seu sorriso único, seu olhar que encanta e sua voz que traz alegria ao meu coração. Companheira fiel e única, que está sempre ao meu lado em todas as situações.

Dedico também o livro à minha família que está lá no interior de Minas Gerais juntamente com minha linda sobrinha Louanne e sua irmã e minha afilhada Fernanda.

Segue um agradecimento sem medidas aqui ao Rodrigo Sasaki (http://cv.rodrigosasaki.com) que me ajudou no decorrer desse livro com revisões em textos e códigos. Ter um profissional de alto calibre como ele ajudando na produção de um livro é de incomensurável alegria. Sou grato a Deus por ter colocado em meu caminho pessoa tão boa, sábia e sempre disposta a ajudar.

E por último, mas não menos importante, dedico esse livro à minha querida irmã Louise, que sempre briga comigo. [=

Sobre o livro

A JPA é um framework que vem ganhando mais espaço no mercado a cada dia que se passa.

Veremos nesse livro diversos conceitos e dicas de utilizações de diversos recursos que a JPA oferece. Esse livro é ideal para quem já entende o conceito do framework e já sabe fazer um "hello world".

Ao final desse livro um desenvolvedor JPA já estará apto a modelar, desenvolver e resolver diversos problemas que podem acontecer ao se trabalhar com JPA.

Sumário

Capítulo 1

Introdução

Diversas vezes, ao trabalhar com JDBC, nos deparamos com rotinas chatas e entediantes ao escrever comandos SQL. Como fazer diversos *whiles* para popular um objeto, erro de sintaxes ao escrever um insert e até mesmo problemas de relacionamento de chaves ao fazer um *delete*.

Veja o código de JDBC a seguir:

```
private Usuario buscaUsuario(int id) throws SQLException {
    Connection conexaoComBanco = null;
    PreparedStatement preparedStatement = null;

    Usuario usuarioEncontrado = null;

    String consulta = "SELECT
                            EMAIL,
                            NOME
                        FROM
                            USUARIO
```

```
                              WHERE
                                 ID = ?";

    try {
        // busca a conexão de algum lugar válido
        conexaoComBanco = getConexaoComBanco();

        // prepara o objeto da conexão
        preparedStatement = conexaoComBanco.prepareStatement(consulta);
        preparedStatement.setInt(1, id);

        // executa a consulta
        ResultSet rs = preparedStatement.executeQuery();

        usuarioEncontrado = new Usuario();

        while (rs.next()) {
            String email = rs.getString("EMAIL");
            String nome = rs.getString("NOME");


            usuarioEncontrado.setNome(nome);
            usuarioEncontrado.setId(id);
            usuarioEncontrado.setEmail(email);
        }
    } finally {
        if (preparedStatement != null) {
            preparedStatement.close();
        }

        if (conexaoComBanco != null) {
            conexaoComBanco.close();
        }
    }

    return usuarioEncontrado;
}
```

O código que acabamos de ver utiliza o JDBC puro para realizar a conexão com o banco de dados e realizar uma consulta. Note como é verboso, necessita de diversas ações para realizar apenas a consulta que volta ou um ou nenhum resultado. Caso

fosse uma lista, teríamos que ter o cuidado de criar uma lista e populá-la. Imagine também se cada usuário tivesse um log de atividades. Seria necessário fazer outra consulta para trazer essa informação, ou utilizar um JOIN, complicando ainda mais a simples tarefa de buscar uma informação no banco de dados.

Para evitar toda essa verbosidade do JDBC, surgiu a JPA, uma ferramenta muito poderosa que pode aumentar consideravelmente o tempo de desenvolvimento da equipe, além de facilitar muito a implementação do código que manipula o banco de dados. Quando corretamente aplicado, a JPA se torna uma ferramenta que ajuda em todas as funções de um *CRUD* (*Create*, *Read*, *Update* and *Delete*), além de contribuir com diversas otimizações de performance e consistência de dados.

É nesse exato momento que podemos entender por que as siglas *ORM* são tão faladas. **O**bject **R**elational **M**apping (*ORM*) quer dizer basicamente: "transformar as informações de um banco de dados, que estão no modelo relacional para classes Java, no paradigma Orientado a Objetos de um modo fácil".

Um framework *ORM* vai ajudá-lo na hora de realizar consultas, manipular dados e até mesmo a retirar relatórios complexos. A verbosidade de um SQL com JDBC não será necessária, nem mesmo os incansáveis loops que fazemos para popular objetos. Tudo isso já está pronto. Só precisamos saber usar.

Outra vantagem que veremos é a portabilidade da aplicação entre banco de dados, tratamento de acesso simultâneo, acesso às funções, dentre outras mais.

O código de busca do usuário pelo `id` que acabamos de ver poderia ser resumido para:

```
private Usuario buscaUsuario(int id){
    EntityManager entityManager = getEntityManager();
    Usuario usuario = entityManager.find(Usuario.class, id);
    entityManager.close();
}
```

Note quão simples e prático ficou nosso código. E para salvar uma informação no banco de dados? Será que teremos que fazer muito código? Na prática, não é necessário criar "queries" longas e tediosas. Bastaria fazer:

```
private void salvarUsuario(Usuario usuario) {
    EntityManager entityManager = getEntityManager();
    entityManager.getTransaction().begin();
    entityManager.persist(usuario);
    entityManager.getTransaction().commit();
```

```
    entityManager.close();
}
```

É possível perceber que o contato com o banco de dados ficou mais simples e de fácil utilização. Rotinas que incluíam escrever tediosos e verbosos comandos SQL agora viraram apenas chamadas a uma API simples. É claro que a vantagem não fica só aí. Vai muito mais além e vamos explorar essas funcionalidades durante o livro.

Veremos desde os conceitos básicos da JPA até os avançados, falando de diversos detalhes e boas práticas, em capítulos curtos, cada um focado em um recurso específico, dessa forma, você pode ler os capítulos de forma isolada e usá-lo como um guia de referência quando precisar.

Capítulo 2

Como escolher uma implementação e as configurações da JPA

2.1 Escolha uma implementação

A JPA nada mais é do que um conjunto de regras, normas e interfaces para definir um comportamento também conhecido como especificação. No mundo Java temos diversas outras como JSF, EJB, JAAS e muito mais. Toda especificação precisa de uma implementação para que possa ser usada num projeto. No caso da JPA, a implementação mais famosa e utilizada no mercado é o Hibernate, mas existem mais no mercado como OpenJPA, EclipseLink, Batoo e outras. Cada implementação tem suas vantagens e desvantagens na hora da utilização.

O Hibernate contém diversas funcionalidades como Engenharia Reversa, Exportação de Schema facilitado e anotações próprias para simplificar a utilização da API. Por outro lado, ao utilizar o Hibernate, será necessário adicionar diversas dependências para o seu correto funcionamento. A ideia do Batoo, por exemplo, é o foco na

performance. Ele tem um número reduzido de dependências, mas tem um desempenho otimizado. Já da OpenJPA é possível dizer que sua simplicidade e facilidade de usar os fazem bem próximos da API da JPA; ao estudar a JPA facilmente poderia se entender a OpenJPA e suas anotações extras.

Uma implementação da JPA deve satisfazer todas as interfaces determinadas pela especificação. Pode acontecer de uma implementação ter 80% de todas as interfaces implementadas e ainda assim lançar uma versão para utilização; fica a critério de cada uma garantir que tudo esteja corretamente implementado. Caso não esteja, usuários escolherão não utilizar. Para implementar a especificação, é necessário implementar as interfaces que estão no pacote `javax.persistence.*` , que é justamente o papel do Hibernate, EclipseLink etc. Mesmo uma implementação incompleta pode ser considerada válida. Na versão 3.x do Hibernate, caso um desenvolvedor adicionasse a anotação `@NamedNativeQuery`, que permite que uma *query* nativa (com a sintaxe do banco de dados) fosse declarada em uma *entity* (entidade), uma exceção seria lançada pelo Hibernate informando que essa funcionalidade ainda não estava implementada.

Além das interfaces implementadas da JPA, cada implementação pode ter as suas próprias anotações/classes para adicionar mais comportamentos, indo além do que prevê a especificação. O Hibernate, por exemplo, tem a anotação `org.hibernate.annotations.ForeignKey`, que consegue determinar o nome de uma chave *foreign key* (chave estrangeira), algo que não existe na JPA, e que quando utilizada pode fazer com que sua aplicação não seja mais portável para outra implementação.

Em contrapartida, a vantagem de ter um projeto usando apenas anotações da JPA é que ela se torna automaticamente portável, possibilitando a troca de Hibernate para o EclipseLink, por exemplo, uma mera questão de alterar o provider (uma simples configuração no `persistence.xml`, que veremos em breve no capítulo 2.2) e claro, a troca do JAR da implementação.

Mas será que é comum eu precisar trocar de implementação? Uma troca poderia acontecer quando a implementação utilizada tem algum bug que não foi resolvido e não tem nenhum *workaround* disponível. Um outro exemplo seria para realizar testes com uma nova implementação que apareceu no mercado.

O Batoo, um dos mais recentes dos que foram citados aqui, tem a proposta de melhorar o desempenho. Há inclusive um estudo em que o famoso blog HighScalability, focado em escalabilidade e performance de software, faz um *benchmark* comparando as implementações da JPA. É apontado que o Batoo pode ser

até 15x mais rápido do que o Hibernate (http://highscalability.com/blog/2012/10/9/batoo-jpa-the-new-jpa-implementation-that-runs-over-15-times.html) . Essa pesquisa foi feita medindo as funções de inserir, alterar, excluir, consulta por JPQL e por Criteria. Vale a pena olhá-la e julgar os critérios nela apresentados.

A vantagem da JPA é que para testar o Batoo, ou qualquer outra implementação, não seriam necessários mais do que 5 minutos para configurar a aplicação.

A desvantagem em utilizar uma aplicação com anotações da JPA apenas é perder as facilidades que a implementação pode fornecer. A anotação `@ForeignKey`, por exemplo, poderia ser adotada apenas para quem utiliza o Hibernate.

```
public class Pessoa{

  @OneToOne
  @JoinColumn(name = "CREATE_BY")
  @ForeignKey(name="FK_USUARIO") // anotação do hibernate
  protected Usuario usuario;

  // ... outras coisas

}
```

É possível ver no código anterior que agora a chave estrangeira terá um nome determinado pela aplicação. Para quem utilizar a JPA 2.1, a anotação `javax.persistence.ForeignKey` estará incorporada, fazendo com que a do Hibernate seja *deprecated*.

É preciso sempre ter em mente o seguinte: se a API nativa JPA não atende às suas necessidades a solução seria procurar alguma implementação que tenha as funções de que você necessita. É uma via de duas mãos, pois o desenvolvedor terá acesso a uma implementação, mas, ao tentar mudá-la muito código terá que ser analisado e refeito.

2.2 Como compor meu persistence.xml?

O arquivo `persistence.xml` é o ponto de partida de qualquer aplicação que use a JPA. Nele ficarão configurações responsáveis por informar com qual banco de dados a aplicação deve se conectar, configurações específicas da implementação, entidades a serem utilizadas, arquivos de configurações e outros.

Esse arquivo deve ficar na pasta `META-INF` do seu projeto, ou seja, na raiz dos pacotes onde se encontram as classes. A imagem 2.1 mostra onde deveria ficar o arquivo utilizando a IDE Eclipse.

> src 633
com 633
page02 631
> page04 633
META-INF 633
persistence.xml 633

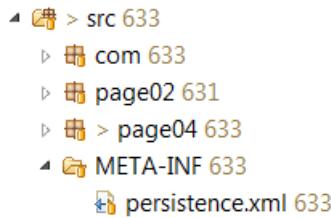



Figura 2.1: persistence.xml em projeto Eclipse

A imagem 2.2 mostra onde deveria ficar o arquivo utilizando Maven.

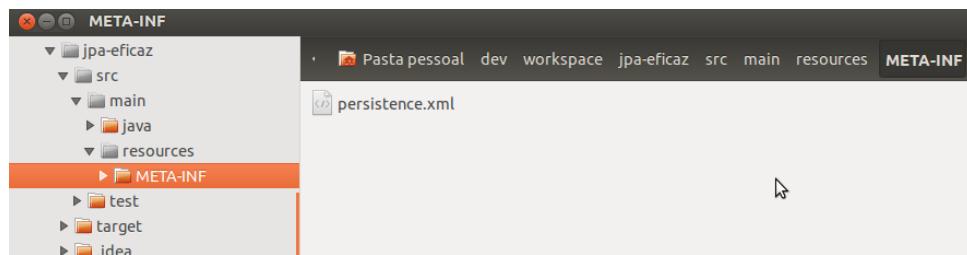



Figura 2.2: persistence.xml em projeto Maven

O arquivo deve estar na raiz das classes dentro da pasta `META-INF`. A imagem 2.3 mostra um JAR que acabou de ser gerado e mostra onde está a pasta `META-INF`. Note que ela se encontra no mesmo nível do pacote `com`, que contém as classes de negócio.

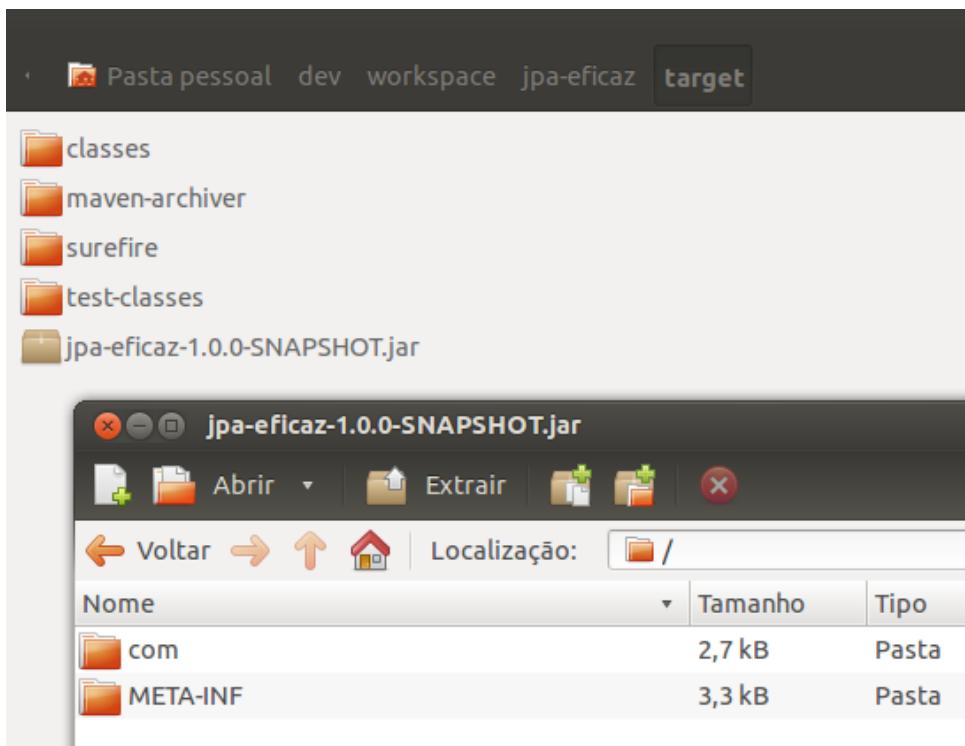



Figura 2.3: persistence.xml em projeto dentro do Jar gerado

Vamos ver agora como um arquivo `persistence.xml` pode ser configurado internamente.

```
<!-- persistence.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
   xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
         http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

</persistence>
```

No código do arquivo `persistence.xml` é possível ver quais são as declarações válidas para utilizar a JPA 2.0. Como saber que é a versão 2.0 da JPA? Basta olhar o atributo `persistence version="2.0"`. É justamente essa con-

figuração que determina a versão da JPA juntamente com o *XSD* utilizado http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd . A tag `persistence` conterá as configurações do projeto, vejamos algumas a seguir:

```
<!-- persistence.xml // declarações omitidas-->
<persistence-unit name="JPA_EFICAZ_HIBERNATE"
                  transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>

    <class>com.jpa.eficaz.model.Musica</class>

  <exclude-unlisted-classes>
    true
  </exclude-unlisted-classes>

    <properties>
        <property name="javax.persistence.jdbc.driver"
                  value="org.hsqldb.jdbcDriver"/>

        <property name="javax.persistence.jdbc.url"
                  value="jdbc:hsqldb:mem:hibernate"/>

        <property name="javax.persistence.jdbc.user"
                  value="sa"/>

        <property name="javax.persistence.jdbc.password"
                  value="senha"/>

        <!-- outras propriedades -->
    </properties>
</persistence-unit>
```

Veja que o código do arquivo `persistence.xml` agora tem diversas propriedades e configurações. `<persistence-unit>` é o universo que a JPA utilizará; nele estarão configurações do banco de dados, arquivos de mapeamentos adicionais, qual implementação usamos, as classes presentes no projeto e outras; `name` define o nome do `persistence-unit` utilizado, e é possível ter mais um `persistence-unit` dentro do mesmo arquivo.

Junto com a declaração do nome do `persistence-unit` é possível ver o atributo `transaction-type`, que é utilizado para indicar à JPA se a transação será

local (RESOURCE_LOCAL) ou controlada pelo servidor (JTA). O exemplo que vimos é do tipo local, a transação será controlada pela aplicação, e em breve veremos um persistence.xml com configuração de JTA.

<provider> indica qual implementação vamos usar. O valor utilizado no exemplo foi a implementação do Hibernate. Caso fosse EclipseLink bastaria utilizar: org.eclipse.persistence.jpa.PersistenceProvider; para a OpenJPA org.apache.openjpa.persistence.PersistenceProviderImpl; para o Batoo seria utilizado %org.batoo.jpa.core.BatooPersistenceProvider. Essa configuração aponta para uma classe dentro do JAR da implementação utilizada. Como visto no capítulo *Escolha uma implementação* (2.1), aqui tem que ser alterado para que haja a troca de implementação.

<class> mapeia quais entidades estarão presentes no "persistence unit". É possível ter entidades diferentes dentro de cada "persistence unit". Uma aplicação Java EE não é obrigada a declarar a tag <class> no persistence.xml, mas para uma aplicação Java SE, algumas implementações podem gerar erro se a tag não for encontrada. Esse erro acontece porque quando estamos trabalhando em um ambiente Java EE, o próprio servidor procurará pelas entidades em seu projeto, já quando a aplicação é utilizada de modo Java SE esse passo inicial pode não existir.

<exclude-unlisted-classes> indica que a aplicação só aceitará as entidades definidas através da tag class dentro do "persistence unit", ignorando caso haja alguma entidade extra na aplicação que não tenha sido listada ali. É uma abordagem muito utilizada quando temos mais de um "persistence unit" declarados no mesmo persistence.xml.

<properties> são as configurações que a implementação utilizará. É exatamente ali que se colocam senha, URL de acesso e outros valores. javax.persistence.jdbc.driver indica qual o Driver JDBC será utilizado na conexão com o banco de dados. javax.persistence.jdbc.url recebe como parâmetro a URL do banco de dados a se conectar. javax.persistence.jdbc.user indica o usuário com permissão para acessar o banco de dados. javax.persistence.jdbc.password a senha do usuário que irá acessar o banco de dados.

Para configurar sua aplicação para utilizar a transação do tipo JTA é simples. Basta adicionar duas configurações:

```
<persistence-unit name="PERSISTENCE_JTA" transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>java:comp/env/jdbc/sgiDS</jta-data-source>
```

```
    ...
</persistence-unit>
```

O código visto mostra como configurar o `persistence unit` do tipo `JTA`. Uma nova tag apareceu chamada `<jta-data-source>`. Ela define qual o `datasource` utilizado pelo servidor.

> ### Erro com "persistence unit" local no JBoss
>
> Pode acontecer que, ao declarar um "persistence unit" do tipo `RESOURCE_LOCAL` no JBoss (versões mais antigas), ele exiba o seguinte erro: "*You have not defined a non-jta-data-source for a `RESOURCE_LOCAL` enabled persistence context named*". Esse erro acontece pois o JBoss precisa que a cada `persistence-unit` declarado exista um datasource associado. Independente do tipo da transação (`RESOURCE_LOCAL` ou `JTA`), para o JBoss, será sempre necessário declarar um datasource.
>
> A solução é simples, mas às vezes difícil de se encontrar. Basta declarar um datasource local.
>
> ```
> <non-jta-data-source>
>   DATA_SOURCE_DA_APLICACAO
> </non-jta-data-source>
> ```
>
> O datasource declarado não será referenciado em lugar nenhum na hora de sua utilização, é apenas para controle do JBoss.

Para finalizar sobre `persistence.xml` vejamos como utilizar e para que serviria declarar mais de um `persistence unit` no mesmo arquivo.

```
<!-- dois_persistences.xml -->
<persistence-unit name="PU_ORACLE" transaction-type="JTA">
    ...
    <class>com.jpa-eficaz.Pessoa</class>
    <exclude-unlisted-classes>true</exclude-unlisted-classes>
    ...
</persistence-unit>

<persistence-unit name="PU_SQLSERVER" transaction-type="JTA">
```

```
    ...
    <class>com.jpa-eficaz.Carro</class>
    <exclude-unlisted-classes>true</exclude-unlisted-classes>
    ...
</persistence-unit>
```

A primeira utilidade é ter dois bancos diferentes na mesma aplicação. No exemplo do arquivo `dois_persistences.xml` é possível ver que temos uma classe que utiliza um banco de dados e outra classe que utiliza um outro banco completamente diferente.

O arquivo `persistence.xml` é fácil, simples de utilizar, prático para dar manutenção e evita códigos complexos que fariam a mesma tarefa. E a melhor parte é que, uma vez configurado, não é necessário sua edição regularmente. Isso geralmente só é necessário quando grandes refactorings acontecem ou quando alguma configuração será modificada.

2.3 Configurando a aplicação através de XML

As anotações são realmente úteis e facilitam no desenvolvimento do dia a dia. Infelizmente elas não ajudam em todos os casos, e são nesses exatos momentos que o conhecimento do XML da JPA vem ajudar.

Um bom exemplo seria uma aplicação que se conecta a dois bancos de dados diferentes. Imagine que a entity `Pessoa` se encontra apenas no database A, mas a entity `Log` se encontra em outro banco. Se fosse usado a configuração apenas por anotação, a JPA entenderia que a entidade `Pessoa` deveria estar nos dois databases e com isso geraria uma mensagem de erro.

São por esses e outros motivos que é possível configurar uma aplicação totalmente por xml. Veremos aqui uma introdução com as configurações que podem ser mais utilizadas em um momento de aperto.

O persistence XML utilizado é o seguinte:

```
<persistence-unit name="PU_BANCO_A" transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>

    <class>com.uaihebert.model.Pessoa</class>
    <exclude-unlisted-classes>true</exclude-unlisted-classes>

    <properties>
```

```
        <!-- configuracoes -->
    </properties>
</persistence-unit>

<persistence-unit name="PU_BANCO_B" transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>

    <class>com.uaihebert.model.Log</class>
    <exclude-unlisted-classes>true</exclude-unlisted-classes>

    <properties>
        <!-- configuracoes -->
    </properties>
</persistence-unit>
```

Note no código que temos dois *Persistence Unit* sendo utilizados só que ambos estão em bancos de dados diferentes. É possível perceber que à medida que a quantidade de entidades forem aumentando, maior esse arquivo XML vai ficar, e pior ficará a leitura. Para evitar esse problema podemos enviar essas configurações para um arquivo à parte:

```
<persistence-unit name="PU_BANCO_A" transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>

    <mapping-file>orm_A.xml</mapping-file>
    <exclude-unlisted-classes>true</exclude-unlisted-classes>

    <properties>
        <!-- configuracoes -->
    </properties>
</persistence-unit>

<persistence-unit name="PU_BANCO_B" transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>

    <mapping-file>orm_B.xml</mapping-file>
    <exclude-unlisted-classes>true</exclude-unlisted-classes>

    <properties>
        <!-- configuracoes -->
    </properties>
```

```
</persistence-unit>
```

Agora existem arquivos com as entidades do banco de dados, o que facilita a leitura e diminui o tamanho do arquivo `persistence.xml`. E o arquivo de mapeamento ficaria algo como:

```
<!-- orm_A.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings
        xmlns="http://java.sun.com/xml/ns/persistence/orm"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
        http://java.sun.com/xml/ns/persistence/orm_2_0.xsd"
        version="2.0">

    <entity class="com.uaihebert.model.Pessoa"/>

</entity-mappings>


<!-- orm_B.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings
        xmlns="http://java.sun.com/xml/ns/persistence/orm"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
        http://java.sun.com/xml/ns/persistence/orm_2_0.xsd"
        version="2.0">

    <entity class="com.uaihebert.model.Log"/>

</entity-mappings>
```

Apenas vendo o código anterior alguém pode pensar: "uau, muito *XML* pra pouca coisa". É nessa hora que vamos ver outra grande vantagem do *XML* utilizado com a JPA. É possível sobrescrever as anotações de uma entity através do XML. Imagine o seguinte, o deploy foi realizado em uma aplicação que é crítica. A entidade `Pessoa` está assim:

```
@Entity
public class Pessoa {
```

```
    @Id
    private int id;
    private String nome;
    // outras coisas
}
```

Só que após o artefato ir para produção aparece uma mensagem de erro falando que a coluna `nome` não foi encontrada na tabela `Pessoa`. Ao olhar na tabela foi possível perceber que a coluna se chama `nome_pessoa`. Imagine que o artefato leva 15 minutos para seu *upload* no servidor terminar. Um novo deploy atrasaria todo o processo e existem casos em que o *rollback* não é possível, pois houve drop de colunas no banco de dados. Se a aplicação utilizar *XML*, bastaria configurar o arquivo `orm_A.xml` como a seguir:

```
<entity class="com.uaihebert.model.Pessoa">
<attributes>
    <basic name="nome">
        <column name="nome_pessoa"/>
    </basic>
</attributes>
</entity>
```

Alterar esse *XML* seria simples e rápido. Após a alteração bastaria fazer o start novamente do servidor que tudo funcionaria sem problemas, e um outro deploy poderia ser agendado para o outro dia. Outro cenário onde *XML* pode fazer diferença é justamente se a coluna `nome` da entity `Pessoa` tivesse nome diferentes em clientes diferentes.

Imagine a situação em que a entity `Pessoa` da aplicação é utilizada em 3 clientes diferentes e cada um com seu banco em específico. Só que no Cliente A temos a coluna chamada de `nome` na tabela `Pessoa`, Cliente B como `nome_pessoa` e Cliente C `nome_txt`. É nesse caso que poderíamos ter um único `persistence.xml` mas na hora de gerar o pacote manipular qual arquivo de mapeamento seria o escolhido. O `persistence.xml` ficaria:

```
<persistence-unit name="PU" transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>

    <mapping-file>orm.xml</mapping-file>
    <exclude-unlisted-classes>true</exclude-unlisted-classes>
```

```
    <properties>
        <!-- configuracoes -->
    </properties>
</persistence-unit>
```

Mas na hora do deploy é que o arquivo `orm.xml` seria escolhido (via ant, maven, manualmente). E os arquivos de mapeamento seriam parecidos com:

```
<!-- Cliente A orm.xml -->
<entity class="com.uaihebert.model.Pessoa">
<attributes>
    <basic name="nome">
        <column name="nome_pessoa"/>
    </basic>
</attributes>
</entity>

<!-- Cliente B orm.xml -->
<entity class="com.uaihebert.model.Pessoa">
<attributes>
    <basic name="nome">
        <column name="nome_pessoa"/>
    </basic>
</attributes>
</entity>

<!-- Cliente C orm.xml -->
<entity class="com.uaihebert.model.Pessoa">
<attributes>
    <basic name="nome">
        <column name="nome_txt"/>
    </basic>
</attributes>
</entity>
```

Note como é prático o XML nesses casos. É possível configurar o *XML* com relacionamentos entre entidades, consultas, geração de ID etc. E por último, é possível também sobrescrever as consultas que já existem, ou até mesmo criar consultas. Veja agora uma pequena alteração que será realizada no `persistence.xml`:

```
<persistence-unit name="PUA" transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
```

```
    <mapping-file>orm.xml</mapping-file>
    <mapping-file>pessoa.xml</mapping-file>
    <exclude-unlisted-classes>true</exclude-unlisted-classes>

    <properties>
        <!-- configuracoes -->
    </properties>
</persistence-unit>
```

Note que agora é possível ver que existe um arquivo chamado `pessoa.xml`, mas o interessante é que esse arquivo não tem as definições da entity `Pessoa`, mas sim apenas consultas:

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings
        xmlns="http://java.sun.com/xml/ns/persistence/orm"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
        http://java.sun.com/xml/ns/persistence/orm_2_0.xsd"
        version="2.0">

    <named-query name="Pessoa.ListAll">
        <query>select p from Pessoa p</query>
    </named-query>

</entity-mappings>
```

Desse modo é possível descrever toda as queries do sistema nos arquivos XML, o que pode ser uma boa alternativa para evitar que a entidade não fique poluída com muitas consultas. Esse arquivo também pode ser utilizado para sobrescrever consultas que subiram erradas nas anotações.

2.4 Como conseguir um EntityManager

Pode parecer algo simples, mas muitas pessoas têm dúvidas sobre como conseguir uma instância de um `EntityManager`.

É possível controlar uma transação manualmente ou deixar que o servidor faça isso por nós. Veremos a seguir ambas as práticas e os cuidados que precisamos ter.

2.4.1 Controlando a transação manualmente

Essa prática é a mais comum de se encontrar em tutoriais, quando temos um programa *standalone* rodando a partir de um método `main` e uma ação é executada. Controlar a transação manualmente pode ser feito também em uma aplicação web, mas com muito cuidado.

Veja como ficaria o código em uma aplicação Java SE:

```
public class Main {
   public static void main(String[] args) throws Exception {
       EntityManagerFactory entityManagerFactory =
      Persistence.createEntityManagerFactory("MeuPU");

      EntityManager entityManager =
         entityManagerFactory.createEntityManager();

      entityManager.getTransaction().begin();

      // faz algo

      entityManager.getTransaction().commit();
      entityManager.close;
   }
}
```

Vendo esse código mais de perto é bom destacar o método `createEntityManagerFactory("MeuPU")`. Com esse comando, a JPA procurará pelo arquivo `persistence.xml`, e fará a leitura das entidades mapeadas, validação de conexão com o banco de dados e, caso configurado, a JPA também poderá validar se o *schema* do banco de dados está correto, criar tabelas, criar sequences etc.

Note que a ação de criar um `EntityManagerFactory` é uma operação muito custosa e não vale a pena repeti-la diversas vezes. Nesse caso o ideal é deixar o `EntityManagerFactory` como `static`, ou seja, uma instância para a aplicação inteira.

Você pode até se perguntar: "E com relação a acesso simultâneo? Como ele se comporta em um ambiente *muti-thread*?" O `EntityManagerFactory` é *thread-safe*, em outras palavras, ele não mandará um `EntityManager` para uma outra *thread* de outro usuário. Com isso é seguro ter sempre o `EntityManagerFactory` como estático. Alterando um pouco a classe anterior, veja como está o código agora:

```
public class Main {
   private static EntityManagerFactory entityManagerFactory =
           Persistence.createEntityManagerFactory("MeuPU");

   public static void main(String[] args) throws Exception {
       EntityManager entityManager =
                        entityManagerFactory.createEntityManager();
       entityManager.getTransaction().begin();
       // faz algo
       entityManager.getTransaction().commit();
       entityManager.close();
   }
}
```

Note que temos um `EntityManagerFactory` fixo agora e o único objeto a chamar o método close é o `EntityManager`. Entretanto em uma aplicação profissional não fica bom chamar uma `EntityManager` de uma classe com um método `main`.

É muito comum encontrar pela internet uma classe chamada de `JpaUtil`. Ela seria utilizada em diversos locais do sistema. `JpaUtil` não é um *pattern* oficializado, é apenas um modo de diminuir o acoplamento da aplicação. Seria algo como:

```
public final class JpaUtil {

   private static final String PERSISTENCE_UNIT = "MeuPU";

   private static ThreadLocal<EntityManager> threadEntityManager =
                                    new ThreadLocal<EntityManager>();

   private static EntityManagerFactory entityManagerFactory;

   private JpaUtil() {
   }

   public static EntityManager getEntityManager() {
       if (entityManagerFactory == null) {
           entityManagerFactory =
             Persistence.createEntityManagerFactory(PERSISTENCE_UNIT);
       }

       EntityManager entityManager = threadEntityManager.get();
```

```
        if (entityManager == null || !entityManager.isOpen()) {
            entityManager = entityManagerFactory.createEntityManager();
            JpaUtil.threadEntityManager.set(entityManager);
        }

        return entityManager;
    }

    public static void closeEntityManager() {
        EntityManager em = threadEntityManager.get();

        if (em != null) {

            EntityTransaction transaction = em.getTransaction();

            if (transaction.isActive()) {
                transaction.commit();
            }

            em.close();

            threadEntityManager.set(null);
        }
    }

    public static void closeEntityManagerFactory() {
        closeEntityManager();
        entityManagerFactory.close();
    }
}
```

Como é possível ver na classe anterior, esse é um padrão que pode vir a ser seguido, mas existem diversas outras melhorias que podem ser realizadas. A única novidade que podemos ver aqui é a classe `ThreadLocal<EntityManager>`, que serve para salvar uma instância de determinada classe por *thread*. Com isso, é possível utilizar seguramente o mesmo `EntityManager` na thread iniciada, e não enviar o `EntityManager` para o usuário errado.

Ao utilizar a abordagem de controle manual da transação é necessário ter muito cuidado com seu início e fim. Ao deixar transações abertas, rapidamente seu sistema

entrará em caos. E aí vem a má notícia: é muito fácil esquecer uma transação aberta.

Nossa classe main agora ficaria mais simples:

```
public class Main {

   public static void main(String[] args) throws Exception {
       EntityManager entityManager = JpaUtil.getEntityManager();
       entityManager.getTransaction().begin();
       // faz algo
       entityManager.getTransaction().commit();
       entityManager.close();
   }
}
```

Veja que o controle do `EntityManagerFactory` simplesmente sumiu da classe que fará a utilização no banco de dados. Desse modo, você começa a aumentar o desacoplamento de suas classes, e poderia, inclusive, deixar o controle do `EntityManager` dentro do `JpaUtil`, mas isso é questão da necessidade. Por enquanto, vamos deixar nossa classe simplesmente fornecendo o `EntityManager` de um modo seguro.

Um outro detalhe interessante sobre o código anterior é: e se entre a abertura e fechamento da transação acontecesse um erro, uma exceção? Note que a transação continuaria aberta. É muito importante ter em mente a importância de sempre fechar a transação. Será sempre necessário um fino controle da utilização da transação da JPA, por exemplo, sempre que capturar uma `Exception`, fazer *rollback*. Veja a seguir:

```
public class Main {

    public static void main(String[] args) throws Exception {
        EntityManager entityManager = JpaUtil.getEntityManager();

        try {
            entityManager.getTransaction().begin();
            // faz algo
            entityManager.getTransaction().commit();
        } catch (Exception e) {
            if(entityManager.isOpen()){
                entityManager.getTransaction().rollback();
            }
```

```
        } finally {
            if(entityManager.isOpen()){
                entityManager.close();
            }
        }
    }
}
```

Para uma aplicação web, controlar manualmente a transação pode ser feito através de um `filter`. Falamos já sobre OpenSessionInView, em que é comum termos um `filter` para controlar a transação. Veja como ficaria o código do `filter` utilizando a nossa nova classe:

```
public class ConnectionFilter implements Filter {

    @Override
    public void destroy() {

    }

    @Override
    public void doFilter(ServletRequest request,
                         ServletResponse response,
                         FilterChain chain)
            throws IOException, ServletException {
        EntityManager entityManager = JpaUtil.getEntityManager();
        try {

            entityManager.getTransaction().begin();

            chain.doFilter(request, response);

            entityManager.getTransaction().commit();
        } catch (Exception e) {
            if(entityManager.isOpen()){
                entityManager.getTransaction().rollback();
            }
        } finally {
            if(entityManager.isOpen()){
                entityManager.close();
            }
```

```
        }

    }

    @Override
    public void init(FilterConfig arg0) throws ServletException {

    }
}
```

É possível ver como o código ficou mais simples e mais fácil de dar manutenção. Só não esqueça de sempre lançar a exceção para o filter ou então um `commit()` poderá acontecer sendo que o comando de `rollback()` é o que deveria ser executado.

2.4.2 Servidor controlando a transação

Essa é a solução mais simples, na qual não precisamos ter o cuidado com a transação ou instanciação do `EntityManagerFactory` nem do `EntityManager`. É preciso primeiramente configurar o `persistence.xml` para utilizar a transação do tipo `JTA`:

```
<!-- persistence.xml // declarações omitidas-->
<persistence-unit name="JPA_EFICAZ_HIBERNATE"
                  transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>

    <jta-data-source>DataSource</jta-data-source>

    <properties>
        <!-- outras propriedades -->
    </properties>
</persistence-unit>
```

Note que no arquivo `persistence.xml` agora a transação é utilizada através do `JTA`. `JTA` significa *Java Transaction API* e é o responsável por abrir, fechar, realizar `commit` e `rollback` nas transações. Essa carga de controle da transação sai dos ombros dos desenvolvedores e fica para o servidor.

Existe no `persistence.xml` a *tag* `<jta-data-source>` que é a configuração que indica ao servidor que ele controlará o acesso ao banco de dados. Essa *tag* indica à JPA como se conectar ao banco de dados.

Não é qualquer servidor que daria suporte a esse tipo de controle de transação. O JBoss, Glassfish e o TomEE são servidores que dão. Esse servidores já vêm com uma implementação da JPA, então não é necessário adicionar essas dependências ao projeto. Infelizmente servidores leves como Tomcat e Jetty não dão esse suporte.

O mais interessante de quando o servidor toma conta da transação é que não é necessário mais executar o comando: `Persistence.createEntityManagerFactory()`. Em algum momento o servidor chamará esse método para que as devidas rotinas sejam executadas.

A utilização do `EntityManager` se dá através de injeção, e para utilizar essa injeção automática é necessário sempre ter em mente uma coisa: **o servidor só injetará o `EntityManager` em classes que ele controla**. Não adianta simplesmente fazer como o seguinte:

```
public class MinhaClasse(){
    @PersistenceContext
    private EntityManager entityManager;
}
```

Note que, aos olhos do servidor, ele não tem a mínima ideia de quem é essa sua classe. É nessa hora que entra em ação o EJB, que é uma classe gerenciada pelo próprio servidor e que fará o controle da transação:

```
@Stateless
public class MinhaClasse(){
        @PersistenceContext
        private EntityManager entityManager;
}
```

Mas e se você não estiver utilizando um servidor que forneça essas funcionalidades ou então não está com tempo de aprender a usar EJB ou até mesmo não gosta do EJB? A solução seria utilizar um framework que fizesse isso por você. Existe por exemplo o Spring (http://www.springsource.org/) que poderia fazer essa injeção e também o Atomikos (http://www.atomikos.com/) . São frameworks bastante utilizados no mercado e que podem e muito facilitar a utilização da transação em seu sistema.

Capítulo 3

Aprenda os detalhes dos mapeamentos de entidades

3.1 Entidades e o seu papel no banco de dados

Como você já sabe, JPA trabalha com objetos Java que refletem o estado dos dados no banco de dados. O ideal é que uma classe possa representar os dados de uma linha de uma ou mais tabelas. Para fazer isso, a JPA criou o conceito de entity (Entidade).

Uma entity refletirá as informações do banco de dados e nada mais é do que uma classe comum, sem nada especial, tendo apenas que seguir 3 simples regras:

- Deve ser anotada com `@Entity`;
- Deve ter um campo definido que representa a chave primária da tabela, anotado com `@Id`;
- Deve ter um construtor público sem parâmetros.

São requisitos simples e fáceis de serem implementados. A entidade `Musica`, que poderia existir em uma aplicação que gerencia *playlists* dos usuários, é um exemplo de como escrever o mínimo para uma entidade:

```
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Musica {

    @Id
    private int id;

    private String nome;

    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
}
```

Sobre o campo definido como `@Id`, pode-se definir que não precisa de método `set`. A ideia da JPA é que o `id` é imutável, então não seria necessário acesso a ele por `set`.

Imagine uma tabela `Pessoa` cujo `@Id` fosse o CPF. Bastaria marcar o campo `private String cpf` com `@Id` para funcionar corretamente. **Toda entity precisa de um campo anotado com @Id**.

Perceba que no código da classe `Musica` em nenhum momento foi citado o nome da tabela no banco de dados. A JPA por padrão procurará uma tabela com o mesmo nome da classe, e o mesmo para todos os campos da classe. A JPA procurará por colunas com o nome `id` e `nome`.

E se o nome da tabela música fosse `tb_music_2421638`? Bastaria então utilizar a anotação `@Table` que já seria possível sobrescrever o valor padrão do nome da classe. Uma grande vantagem da JPA é que, quando falamos de mapeamento de banco de dados, é possível alterar os comportamentos padrões nele previsto. Nome de coluna, nome de tabela, nome de relacionamentos, dentre outros. Veja a seguir como utilizar a anotação `@Table` e `@Column`.

```
import javax.persistence.*;

@Entity
@Table(name = "tb_music_2421638")
public class Musica {

    @Id
    private int id;

    @Basic(optional = false)
    @Column(name = "NM_MUSICA", length = 100, unique = true)
    private String nome;

    @ManyToOne
    private Album album;

    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
}
```

As anotações `@Table`, `@Basic` e `@Column` definem um relacionamento direto entre a classe e o banco de dados, alterando todo os valores padrões que a JPA seguiria. `@Basic` está definindo que o campo é obrigatório, `@Column` está alterando o nome da coluna no banco de dados e seus valores.

Faça o debug das Anotações

Nessa área veremos em detalhes atributos das anotações apresentadas nesse capítulo. As anotações vistas nesses capitulo foram: @Entity, @Table, @Basic e @Column.

Atributos do @Entity

```
@Entity(name = "MinhaEntity")
public class Pessoa{...}
```

A anotação `@Entity` tem o atributo `name` que pode alterar como a entity é

conhecida pela JPA. Esse valor terá influência, por exemplo, na forma de realizar pesquisas no banco de dados através de JPQLs. Uma JPQL da entity anterior seria algo como

```
select p from MinhaEntity p
```

Atributos do @Table

A anotação `@Table` possui o atributo `name` que define o nome da tabela caso ela não siga o mesmo nome da entity; `catalog` indica o catálogo de metadados do banco; `schema`, a qual schema a tabela pertence; `uniqueConstraints` indica quais *Constraints* únicas devem existir na tabela. Esse valor só é verificado se a JPA for criar as tabelas. Veja como fica uma entidade com todas essas configurações.

```
@Table(name = "PESSOA_TB",
       catalog = "db_catalog",
       schema = "table_schema",
       uniqueConstraints={
         @UniqueConstraint(
            columnNames={"codigo", "nome"}
         ),
         @UniqueConstraint(
            columnNames={"outroCampo", "outroNome"}
         )
       }
)

public class Pessoa{...}
```

Atributos do @Basic

Para a anotação `@Basic` temos dois atributos: `optional` e `fetch`. Essa anotação é padrão para todo atributo de uma classe, ou seja, anotar um atributo com @Basic é o mesmo que não anotar. A função dela é poder definir os dois atributos que veremos a seguir.

O atributo `optional` define se o valor pode estar `null` na hora da persistência e `fetch` indica se o conteúdo será carregado juntamente com a *Entity* quando ela for buscada no banco de dados. Veremos mais sobre fetch ainda nesse livro (capítulo 4.4).

```
@Basic(optional = true, fetch = FetchType.EAGER)
```

Atributos do @Column

Das anotações vistas, a que possui mais atributos é a `@Column`, que permite customizar a maneira com que as colunas serão representadas no banco de dados.

- `name`: indica o nome da coluna do banco de dados para aquele atributo;
- `length`: indica o tamanho do campo, geralmente aplicado a campos de texto;
- `unique`: indica se pode haver valores repetidos naquela coluna da tabela;
- `nullable`: indica se a coluna aceita informações `null`;
- `columnDefinition`: permite definir a declaração de como é a coluna, usando a sintaxe específica do banco de dados. O problema de definir essa opção é que a portabilidade entre banco de dados pode ser perdida;
- `insertable` e `updatable`: indica se aquele campo pode ser alterado ou ter valor inserido;
- `precision` e `scale`: servem para tratar números com pontos flutuantes, como `double` e `float`;
- `table`: indica a qual tabela aquela coluna pertence.

```
@Column(
        name = "NM_MUSICA",
        length = 100,
        unique = true,
        nullable = false,
        columnDefinition = "VARCHAR(45)",
        insertable = true,
        updatable = true,
        precision = 2,
        scale = 2,
        table = "outra_tabela")
private String nome;
```

Concluindo

A ideia da JPA já vir com seu comportamento padrão é algo que facilita e muito o trabalho no dia a dia. Uma vez que esse comportamento é conhecido pelo desenvolvedor, seu trabalho rende mais e fica com mais qualidade.

É bom sempre lembrar que, uma vez que necessário, é possível alterar esse comportamento padrão de modo simples e prático.

3.2 Saiba Gerar seu id Automaticamente

Como já dito no capítulo 3.1, toda *Entity* precisa de um `id`. A classe `Musica` mostra como é uma classe apenas com o `id` e um atributo String declarado.

```
@Entity
public class Musica {

    @Id
    private int id;

    private String nome;

    public Musica(){
    }

    public Musica(int id,String nome){
       this.id =  id;
       this.nome =  nome;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }
}
```

Veja no código da classe `Musica` que é necessário definir um `id` antes de persistir o objeto no banco de dados. O código a seguir mostra como fazer isso:

```
Musica musica = new Musica(1, "Breathe Into Me");
entityManager.persist(musica);
```

Para persistir corretamente uma entidade no banco de dados, foi necessário informar qual o `id`. O problema dessa abordagem é que não temos como saber qual `id` já existe no banco de dados.

Poderíamos fazer o controle do `id` manualmente ou então deixar que a JPA fizesse isso por nós. Na abordagem manual, teríamos que realizar uma consulta no banco de dados que nos devolvesse o último `id` gravado e utilizar o resultado para definir o próximo valor.

```
int proximoId = // select max(id) from musica;

proximoId++;

Musica musica = new Musica(proximoId, "Breathe Into Me");

entityManager.persist(musica);
```

Apesar de funcional, existem diversos problemas nessa abordagem, sendo que um dos principais, e que pode invalidar o uso dessa estratégia, é o fato de poder acontecer acessos simultâneos, fazendo com que o número devolvido seja inconsistente. Uma forma mais fácil de resolver esse problema, é usar as estratégias da própria JPA.

A JPA tem 4 modos para gerar automaticamente o `id` de uma entidade: `IDENTITY`, `SEQUENCE`, `TableGenerator` e `AUTO`. Essa configuração é feita através da anotação `@GeneratedValue`, onde será definido o atributo `strategy`.

```
@Id
@GeneratedValue(strategy = ?)
private int id;
```

Estratégia IDENTITY

O tipo de geração `IDENTITY` nada mais é do que o famoso autoincremento do banco de dados. Para as pessoas que trabalham com MySQL ou SQLServer, esse é o padrão. Para definir a entidade com esse tipo de geração de `id` basta fazer como a seguir.

```
import javax.persistence.*;
```

```
@Entity
public class Musica {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

   // outros métodos omitidos
}
```

A geração `IDENTITY` é controlada pelo banco de dados e ele fica responsável por gerar o próximo valor do `id`. A imagem 3.1 explica melhor esse comportamento.

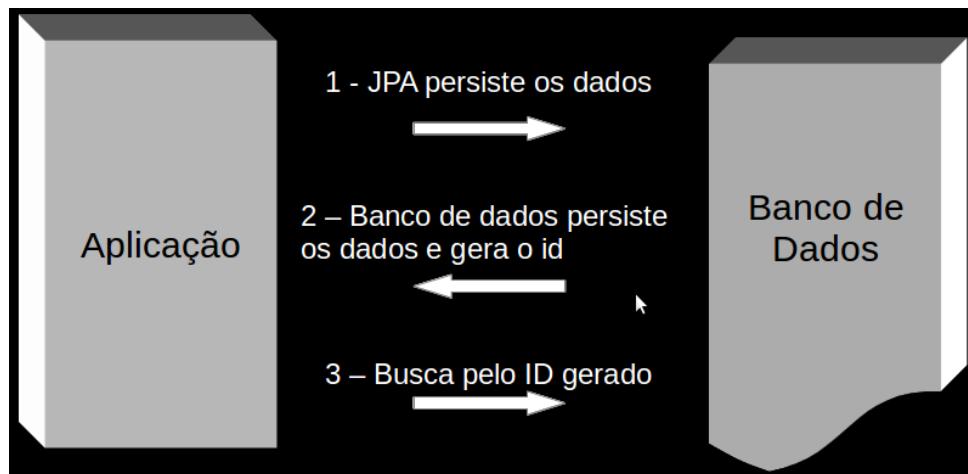



Figura 3.1: `IDENTITY` utilizado pela JPA

Após a persistência do objeto, a JPA deverá buscar pelo `id` recém-gerado. Essa busca poderia gerar uma pequena perda de performance, mas nada alarmante.

Esse tipo de geração não é portável entre os bancos. Sua aplicação não poderia, por exemplo, rodar nos bancos do Postgres ou Oracle utilizando `IDENTITY` para gerar o `id`, já que essa estratégia não é suportada por esses bancos.

Estratégia SEQUENCE

A estratégia `SEQUENCE` utiliza uma rotina que se encontra no banco de dados, que, ao ser chamada, devolve automaticamente o próximo número sem problemas

de concorrência. Esse esquema é o padrão do Postgres e Oracle.

```
import javax.persistence.SequenceGenerator;

@Entity
@SequenceGenerator(name = Musica.SEQUENCE_NAME,
                   sequenceName = Musica.SEQUENCE_NAME,
                   initialValue = 10,
                   allocationSize = 53)
public class Musica {

    public static final String SEQUENCE_NAME = "SEQUENCIA_MUSICA";

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
                    generator = SEQUENCE_NAME)
    private int id;

   // outros métodos omitidos
}
```

A classe `Musica` agora está com a anotação `@SequenceGenerator`. Essa anotação serve para declarar a utilização de uma `SEQUENCE` no banco de dados. O atributo `name` é o nome pelo qual a `SEQUENCE` será conhecida dentro da aplicação. Já o `sequenceName` indica o nome da sequência no banco de dados. O `initialValue` indica o valor inicial da sequência, que no caso do exemplo seria 10, ou seja, quando o primeiro objeto fosse persistido o valor já iria para 11.

Por fim, o `allocationSize` é a quantidade de IDs alocados na memória para ser utilizado e funciona como um cache de valores, que no exemplo anterior alocou 53 números em memória. A cada objeto persistido no banco de dados a JPA utiliza um número desse cache e após utilizar todos os valores disponíveis, ele irá ao banco de dados buscar por mais um bloco de números e dessa forma otimiza o tempo gasto na persistência de objetos.

A anotação `@GeneratedValue` agora define qual `SEQUENCE` será utilizada e também aponta para o nome pelo qual ela será encontrada no banco de dados.

A imagem 3.2 demonstra esse comportamento.

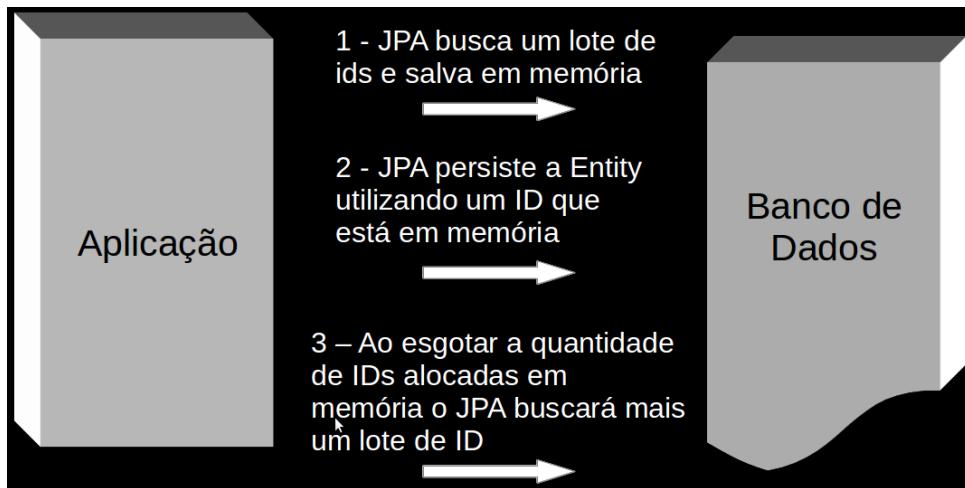



Figura 3.2: `SEQUENCE` utilizada pela JPA

Note na imagem 3.2 que a JPA carregará para a memória um lote de IDs (passo 1). Toda vez que um objeto for persistido, a JPA buscará nesse lote em memória o próximo `id` a ser inserido (passo 2). Uma vez que se esgote o número de IDs armazenados em memória, a JPA alocará em memória um novo lote de `id` (passo 3).

Essa estratégia não permite a portabilidade de uma aplicação para bancos como SQLServer ou MySQL, pois esses bancos não possuem suporte a `SEQUENCE`.

Estratégia TableGenerator

A estratégia `TableGenerator` salva todas as chaves em uma tabela. Essa tabela será composta de duas colunas, sendo que uma indica o nome da tabela e a outra, o valor do `id` atual para aquela tabela. É possível ver na imagem 3.3 como a JPA utilizaria.

TABELA_DE_IDS	
tabela	id_atual
MUSICA	12
FAIXA	33
BANDA	5
COMPOSITOR	2

Figura 3.3: Tabela utilizada pelo `TableGenerator` da JPA

A configuração é feita através da anotação `@TableGenerator`, que exige as definições dos nomes de colunas do nome e valor.

```
import javax.persistence.*

@Entity
@TableGenerator(name= Musica.IDS_TABLE_NAME,
                table="tabela_de_ids",
                pkColumnName="tabela",
                pkColumnValue="musica_id",
                valueColumnName="id_atual")
public class Musica {

    public static final String IDS_TABLE_NAME = "TABELA_DE_IDS";

    @Id
    @GeneratedValue(strategy = GenerationType.TABLE,
                    generator = IDS_TABLE_NAME)
    private int id;

   // outros métodos omitidos
}
```

A anotação `@TableGenerator` indica que o esquema de geração de `id` utilizará uma tabela para armazenar o `id` atual de cada *Entity*. O nome da tabela

com os `ids` é definido pelo atributo `table`. Já o `pkColumnName` é o nome da coluna, da tabela do banco de dados, que armazenará o nome da chave da *Entity*. `pkColumnValue` é o nome que a *Entity* terá salvo na coluna definida no atributo `pkColumnName`, em nosso caso, `musica_id`.

Por sua vez, `valueColumnName` informa a coluna que armazenará o valor atual do `id`.

A imagem 3.3 detalha cada atributo da anotação e sua utilização.

table
pkColumnName
valueColumnName
pkColumnValue

TABELA_DE_IDS	
tabela	id_atual
MUSICA	12
FAIXA	33
BANDA	5
COMPOSITOR	2

Figura 3.4: Tabela utilizada pelo `TableGenerator` da JPA

Basta configurar a anotação `@GeneratedValue` para utilizar a estratégia `TABLE` e apontar qual o `generator`.

A estratégia `@TableGenerator` é a única que permite a portabilidade da aplicação entre bancos de dados. Ela pode ser utilizada com Oracle, MySQL, SQL Server, Postgres e qualquer outro banco.

Os valores `initialValue` e `allocationSize` têm a mesma função que no uso da `SEQUENCE` e com o `allocationSize` é possível otimizar a performance.

Estratégia AUTO

O modo `AUTO` para gerar `id` é o mais simples de todos e, quando utilizado, a JPA ficará encarregada de escolher o modo como será realizada a geração do `id`. Esse é o valor padrão da anotação `@GeneratedValue`.

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private int id;
```

Conclusão

Caso o seu banco só dê suporte a IDENTITY você terá que usar ou IDENTITY ou TABLE_GENERATOR. TABLE_GENERATOR é a única estratégia que pode ser utilizada com qualquer banco de dados. É preciso saber qual o modo de geração de seu banco de dados e utilizar.

Se o seu projeto for ser utilizado por diversos bancos de dados, a questão da chave é um ponto importante para se levar em consideração.

É possível perceber também que o tipo da chave escolhida pode influenciar na performance. É preciso ter sempre em mente qual a melhor escolha para o seu projeto.

3.3 O eterno problema do mapeamento de chaves compostas

Por diversos momentos, um campo apenas não é o suficiente para definir o id de uma entity. Imagine que Musica tenha por chave o nome e a sua duração em segundos. Nesse caso, precisamos de uma **chave composta**.

Anotar dois atributos com @Id não seria a abordagem correta e a JPA exibiria mensagem de erro: *org.hibernate.MappingException: Composite-id class must implement Serializable*. Apesar de a descrição da exceção indicar outra coisa, o que precisamos fazer para definir corretamente chaves compostas através da JPA é utilizar uma classe auxiliar.

Existem duas formas de definir uma chave composta: através das anotações @IdClass ou @EmbeddedId.

Defina chaves compostas em classes separadas com @IdClass

Quando temos uma chave composta e vamos trabalhar com @IdClass, precisamos primeiro indicar uma classe que terá os atributos da chave composta dentro dela. Podemos chamá-la de MusicaId. Vamos indicar que ela é quem terá os IDs de Musica.

```
import javax.persistence.*;

@Entity
@IdClass(MusicaId.class)
public class Musica {
```

```
    @Id
    private int duracaoSegundos;

    @Id
    private String nome;

    // getters e setters
}
```

Note que ainda temos os dois atributos anotados com `@Id` e já sabemos que isso não funciona. Todos os atributos com essa anotação devem estar na classe indicada com `@IdClass`, no caso, a classe `MusicaId`.

```
public class MusicaId implements Serializable {
    private int duracaoSegundos;
    private String nome;

    public MusicaId(){ }

    public MusicaId(int duracaoSegundos, String nome) {
       super();
       this.duracaoSegundos = duracaoSegundos;
       this.nome = nome;
    }

    // implementação do hashCode e equals
}
```

A classe `MusicaId` é bem simples, mas há algumas regras que devem ser seguidas, além de ter os atributos que constituem a chave:

1) A classe deve ter um construtor público sem parâmetros;

2) A classe deve implementar a interface `Serializable`;

3) A classe deve sobrescrever os métodos `hashCode` e `equals`.

```
int duracaoEmSegundos = 196;
String nome = "Breathe Into Me";

MusicaId musicaId = new MusicaId(duracaoEmSegundos, nome);
```

```
Musica musicaSalva = entityManager.find(Musica.class, musicaId);

System.out.println(musicaSalva.getNome());
```

@EmbeddedId

Pode parecer estranho ter todos os atributos do ID dentro da própria *Entity*, para isso existe a solução que vem junto da anotação `@EmbeddedId`. Essa abordagem é utilizada adicionando a classe de ID dentro da própria *Entity*. Veja a seguir como fazer:

```
// Musica.java
import javax.persistence.*;

@Entity
public class Musica {

    @EmbeddedId
    private MusicaId id;

    // getters e setters
}
```

A entity `Musica` agora tem o atributo `id`. A anotação `@EmbeddedId` é utilizada para indicar à JPA que aquela classe contém o `id` da entity.

A classe `MusicaId` continua com a mesma estrutura, com a única diferença de que agora usaremos a anotação `@Embeddable`. Essa anotação fala para a JPA que a classe `MusicaId` fará parte de uma entity como um atributo, como se ela fosse "anexada" a uma outra classe, o que não acontecia com o `IdClass`.

```
import javax.persistence.*;

@Embeddable
public class MusicaId implements Serializable {
    private int duracaoSegundos;
    private String nome;

    public MusicaId(){ }

    public MusicaId(int duracaoSegundos, String nome) {
```

```
        super();
        this.duracaoSegundos = duracaoSegundos;
        this.nome = nome;
    }

    // hashCode e equals
}
```

Mapeie chaves compostas complexas

Uma chave composta complexa possui outra entidade como parte de seu ID. Imagine que a entity `CodigoUnico` deva ser o ID da entity `Musica`.

```
import javax.persistence.*;

@Entity
public class CodigoUnico {

    @Id
    private int id;

    @Temporal(TemporalType.TIMESTAMP)
    private Date dateRegistro;

    private String condigoUnicoHash;
}
```

A entity `CodigoUnico` possui um campo `id`, `Date` e `String` e terá informações de como e quando uma `Musica` foi registrada.

A entity `Musica` agora tem por ID a entity `CodigoUnico` e também deve implementar Serializable. Um relacionamento `OneToOne` foi criado e ele foi anotado com `@Id`. Desse modo, a JPA entenderá que `Musica` terá a mesma chave que `CodigoUnico`.

```
import javax.persistence.*;

@Entity
public class Musica implements Serializable {

    @Id
    @OneToOne
```

```
    @JoinColumn(name = "codigo_id")
    private CodigoUnico codigoUnico;

    // getters e setters
}
```

E se `Musica` agora tivesse outra *Entity* como chave primária? Agora além de `CodigoUnico` a musica terá a entity `Pessoa` (autor) como chave.

```
import javax.persistence.*;

@Entity
public class Pessoa {

    @Id
    private int id;

    private String nome;

    private String nomeArtistico;

    // getters e setters
}
```

Para isso, uma pequena alteração será necessária na entity `Musica` e na classe `MusicaId`.

```
import javax.persistence.*;

@Entity
@IdClass(MusicaId.class)
public class Musica {

    @Id
    @OneToOne
    @JoinColumn(name = "codigo_id")
    private CodigoUnico codigoUnico;

    @Id
    @OneToOne
    @JoinColumn(name = "pessoa_id")
    private Pessoa autor;
```

```
    // getters e setters
}
```

Agora, a entity `Musica` tem dois relacionamentos marcados com `@Id` e está anotada novamente com `@IdClass`. A classe `MusicaId` também foi alterada para apontar para as classes definidas como IDs.

```
import javax.persistence.Embeddable;

public class MusicaId implements Serializable {
    private int codigoUnico;
    private int autor;

    public MusicaId(){ }

    public MusicaId(int codigoUnico, int autor) {
       super();
       this.codigoUnico = codigoUnico;
       this.autor = autor;
    }

    // hashCode e equals
}
```

A classe `MusicaId` tem dois atributos que definem os IDs da classe. Os atributos apontam diretamente para o nome do atributo na *Entity*, por isso se chamam de `autor` e `codigoUnico`.

3.4 Mapeando mais de uma tabela

Em sistemas legados é comum encontrar situações em que duas ou mais tabelas possam representar uma entity. Imagine que a entity `Musica` seja composta da tabela `MUSICA_2372` e da tabela `MUSICA`.

```
@Entity
@Table(name="MUSICA")
@SecondaryTable(
    name="MUSICA_2372",
    pkJoinColumns={@PrimaryKeyJoinColumn(name="MUSICA_ID")}
)
```

```
public class Musica {

    // atributos

}
```

A entity `Musica` está anotada com `@SecondaryTable` que indica qual a outra tabela que contém dados da entity. O atributo `name` informa o nome da outra tabela, enquanto `pkJoinColumns` recebe um array de `@PrimaryKeyJoinColumn` indicando o nome do campo que é a chave primária da tabela.

E se fosse necessário mapear mais tabelas? Imagine agora que a entity `Musica` deverá usar mais uma tabela chamada `INTERNATIONAL_MUSIC`.

```
@Entity
@Table(name="MUSICA")
@SecondaryTables({
    @SecondaryTable(
        name="MUSICA_2372",
        pkJoinColumns={@PrimaryKeyJoinColumn(name="MUSICA_ID")}
    ),

    @SecondaryTable(
        name="INTERNATIONAL_MUSIC",
        pkJoinColumns={@PrimaryKeyJoinColumn(name="MUSICA_ID")}
    )
})
public class Musica {
    // atributos
}
```

A anotação `@SecondaryTables` serve para agrupar as anotações `@SecondaryTable`. Desse modo, é possível apontar o número de tabelas necessárias para compor a entity.

Imagine agora que nas duas tabelas tem um atributo chamado "nome". Como a JPA saberia qual chamar? Em casos como esse, basta apenas utilizar a anotação `@Column` e preencher o atributo `name` com o valor da tabela:

```
@Column(table = "TABELA_X")
private String nome;
```

3.5 Como mapear herança da melhor maneira?

Em uma aplicação Java é normal encontrar herança entre classes, e não seria diferente com entities da JPA. Para trabalhar com hierarquias entre as classes, a JPA fornece diversas alternativas.

Para herança é possível utilizar as estratégias *mapped superclass*, *single table*, *joined* e *table per concrete class*. Cada tipo de herança tem suas vantagens e desvantagens que podem influenciar no tipo de manutenção dos dados, investigação de informações e na estruturação das classes.

3.5.1 Mapped Superclass

Pode acontecer que uma entity herde de uma classe que não seja entity. Imagine que temos a classe `Departamento`.

```
import javax.persistence.MappedSuperclass;

@MappedSuperclass
public abstract class Departamento {

    private String nome;

    public abstract void calcularDespesasDoMes();

    // get e set
}
```

Veja que a classe `Departamento` está anotada com `@MappedSuperclass` que indica que pode ser utilizada em herança de entity, ou seja, toda entity que herdar de `Departamento` deverá ter em sua tabela do banco de dados uma coluna relacionada a `quantidadePortas`.

```
import javax.persistence.*;

@Entity
public class RecursosHumanos extends Departamento {

    @Id
    private long id;
```

```
    @Override
    public void calcularDespesasDoMes () {
        // realiza os cálculos
    }
}
```

A entity `RecursosHumanos` herda da classe `Departamentos` os atributos que lá existirem e é obrigada a sobrescrever os métodos abstratos.

Uma classe anotada com `@MappedSuperclass` não é uma entity, ela não pode ser anotada com `@Entity` ou `@Table`.

Uma `@MappedSuperclass` não é persistida, pois ela não é uma entity. Ela também não pode ser consultada através de *JPQL/HQL*.

Tem-se por boa prática deixar uma `@MappedSuperclass` como abstrata, apenas para definir uma herança. Como esse tipo de classe não pode ser consultada ou persistida não existe motivo para deixá-la como concreta.

Uma `@MappedSuperclass` é ideal para quando se tem uma herança na qual a primeira classe da hierarquia não é persistida e nem consultada diretamente por HQL/JPQL.

3.5.2 SINGLE$_T$*ABLE*

É possível ter toda a herança persistida e com seus dados em uma única tabela. Vamos trabalhar com o modelo de herança visto na imagem 3.5.

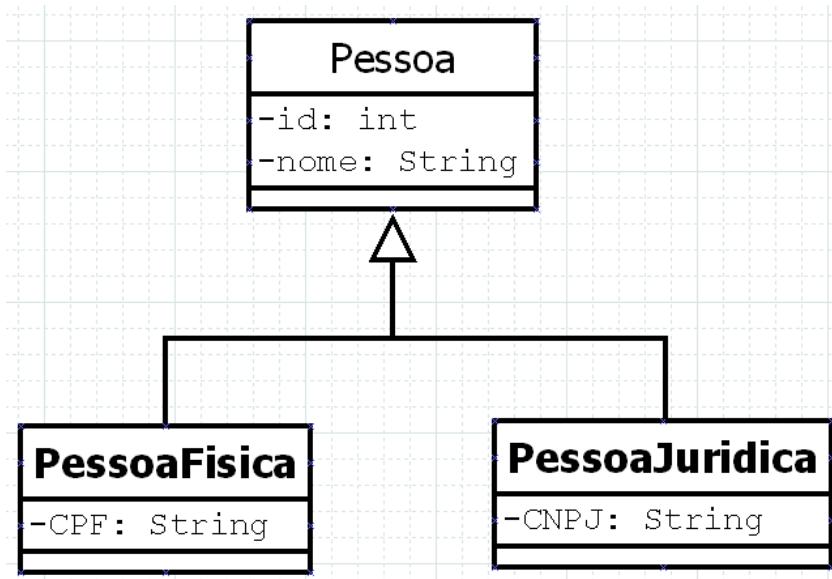



Figura 3.5: Herança a ser utilizada como exemplo

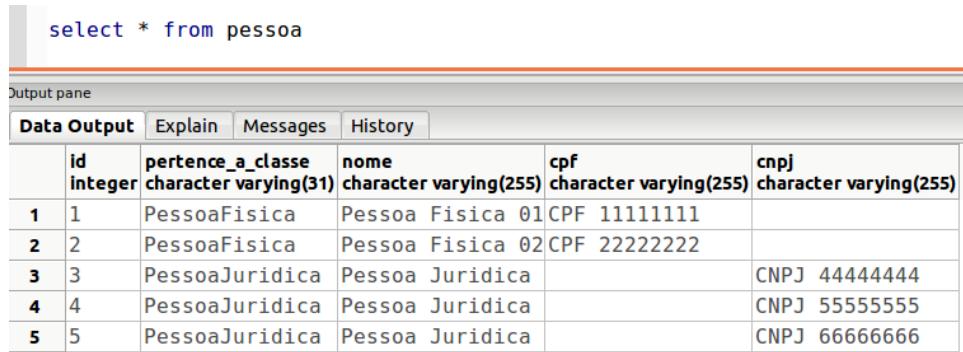
O código de nossa herança é bem simples, uma superclasse e 2 classes herdando.

O tipo de herança *SINGLE_TABLE* tem por característica salvar todas as informações das *Entities* em uma única tabela. Olhando então a herança vista na imagem 3.5 as informações das *Entities "PessoaJuridica"* e *"PessoaFisica"* seriam todas salvas em uma única tabela. O código da classe `Pessoa` mostra como deve ficar esse mapeamento.

```
import javax.persistence.*;

@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "pertence_a_classe")
public abstract class Pessoa {

    @Id
    @GeneratedValue
    private int id;

    private String nome;
```

```
    // get and set
}
```

É possível ver na classe `Pessoa` as anotações `@Inheritance` e `@DiscriminatorColumn`. O tipo de herança `InheritanceType.SINGLE_TABLE` é escolhida no atributo `strategy`. Caso exista uma relação de herança em uma entity e a anotação `@Inheritance` não esteja presente, a JPA adotará a estratégia *SINGLE_TABLE*. `@DiscriminatorColumn` informa qual o nome da coluna que armazenará a entity "dona" de uma determinada linha no banco de dados (daqui a pouco veremos como ficará a tabela no banco de dados, não saia daí).

```
import javax.persistence.*;

@Entity
@DiscriminatorValue("PessoaJuridica")
public class PessoaJuridica extends Pessoa{

    private String CNPJ;
    // outras coisas
}
```

```
import javax.persistence.*;

@Entity
@DiscriminatorValue("PessoaFisica")
public class PessoaFisica extends Pessoa{

    private String CPF;
    // outras coisas
}
```

As *Entities* `PessoaJuridica` e `PessoaFisica` têm em comum a anotação `@DiscriminatorValue` com o valor que irá identificar cada classe na tabela do banco de dados. Para finalizar, olhemos a imagem 3.6, que mostra a tabela das *Entities* já vistas aqui utilizando a herança do tipo *SINGLE_TABLE*.

```
select * from pessoa
```

Output pane

Data Output | Explain | Messages | History

	id integer	pertence_a_classe character varying(31)	nome character varying(255)	cpf character varying(255)	cnpj character varying(255)
1	1	PessoaFisica	Pessoa Fisica 01	CPF 11111111	
2	2	PessoaFisica	Pessoa Fisica 02	CPF 22222222	
3	3	PessoaJuridica	Pessoa Juridica		CNPJ 44444444
4	4	PessoaJuridica	Pessoa Juridica		CNPJ 55555555
5	5	PessoaJuridica	Pessoa Juridica		CNPJ 66666666

Figura 3.6: Estrutura da tabela utilizada por herança SINGLE_TABLE

Na imagem 3.6 é possível ver que existe uma coluna chamada `pertence_a_classe` que foi definida na anotação `@DiscriminatorColumn` encontrada na entity `Pessoa`. Já os valores foram definidos na anotação `@DiscriminatorValue`. Note que cada linha tem definida a qual entity pertence e seus respectivos valores.

É possível definir as vantagens do `SINGLE_TABLE` como:

- Dados Centralizados — os dados estão em uma única tabela, fácil de localizar todos os dados;
- Fácil de entender — um desenvolvedor júnior poderia facilmente analisar os dados, facilita a extração de dados via SQL;
- Boa Performance — tem uma performance boa pois a consulta é realizada em apenas uma tabela. É possível também fazer otimizações como criação de index no banco de dados.

A desvantagem dessa abordagem é que uma entity que herde da classe pai não pode ter campos definidos como `not null`. Imagine que `PessoaFisica` tivesse o campo `double valorValeRefeicao` como `not null`, ao persistir um `PessoaJuridica` uma mensagem de erro do banco de dados seria exibida, pois o campo `valorValeRefeicao` estaria `null`. A solução para esse problema seria deixar no banco de dados aceitando `null` e validar manualmente o campo pedigree. Essa validação poderia ser feita pela anotação `@NotNull` da JSR-305.

3.5.3 JOINED

A estratégia `JOINED` utiliza da abordagem de uma tabela para cada entity da herança. Pequenas alterações serão necessárias nas *Entities* do nosso UML (3.5).

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Pessoa { ... }

@Entity
public class PessoaFisica extends Pessoa { ... }

@Entity
public class PessoaJuridica extends Pessoa { ... }
```

Como é possível ver nas classes `Pessoa`, `PessoaJuridica` e `PessoaFisica`, apenas o tipo herança foi definido na superclasse e nada mais. O tipo `JOINED` tem por característica criar uma tabela por entity, então dessa vez teremos 3 tabelas no banco de dados. Veja nas imagens 3.7, 3.8, 3.9 e 3.10 como fica a estrutura das tabelas.

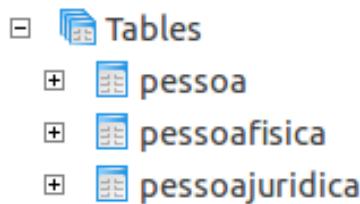



Figura 3.7: Estrutura da tabela utilizada por herança JOINED

```
select * from pessoa
```

id integer	nome character varying(255)
1	Pessoa Fisica 01
2	Pessoa Fisica 02
3	Pessoa Juridica
4	Pessoa Juridica
5	Pessoa Juridica

Figura 3.8: Estrutura da tabela utilizada por herança JOINED

```
select * from pessoafisica
```

id integer	cpf character varying(255)
1	CPF 11111111
2	CPF 22222222

Figura 3.9: Estrutura da tabela utilizada por herança JOINED

```
select * from pessoajuridica
```

ıt pane

ıta Output | Explain | Messages | History

id integer	cnpj character varying(255)
3	CNPJ 44444444
4	CNPJ 55555555
5	CNPJ 66666666

Figura 3.10: Estrutura da tabela utilizada por herança JOINED

`JOINED` trabalha com uma tabela por entity independente de ela ser ou não abstrata.

As vantagens dessa abordagem são:

- Tabela por Entity — essa abordagem coloca cada entity em uma tabela permitindo assim campos `null`;
- Segue modelo OO — as tabelas serão reflexo do OO aplicado nas *Entities*.

As desvantagens seriam:

- Insert mais custoso — um *insert* no banco de dados custaria "mais caro". Para persistir a entity `PessoaFisica` seria necessário realizar *insert* na tabela `pessoafisica` e `pessoa`;
- Alto número de JOINS — quanto maior a hierarquia, maior o número de joins em uma consulta para trazer a entity do banco de dados.

3.5.4 TABLE$_{P}ER_{C}LASS$

`TABLE_PER_CLASS` trabalha com a ideia de uma tabela por classe concreta. Uma alteração será necessária na entity `Pessoa` do nosso UML (3.5).

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Pessoa { ... }
```

```
@Entity
public class PessoaFisica extends Pessoa{ ... }

@Entity
public class PessoaJuridica extends Pessoa{ ... }
```

Na entity `Pessoa`, foi definido o tipo de herança através do enum `InheritanceType.TABLE_PER_CLASS`. Com esse tipo de herança, a JPA utilizará uma entity concreta por tabela. Uma entity abstrata não terá sua tabela própria. Em nosso caso, os atributos existentes na entity `Pessoa` serão encontrados nas tabelas das *Entities* que herdarem da superclasse. Veja nas imagens 3.11, 3.12 e 3.13 como fica a estrutura das tabelas.

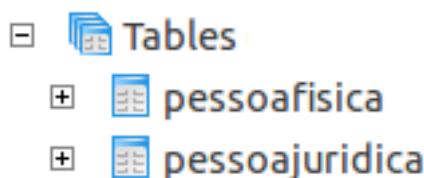



Figura 3.11: Estrutura da tabela utilizada por herança TABLE_PER_CLASS

```
select * from pessoafisica
```

ıt pane

ta Output | Explain | Messages | History

id integer	cpf character varying(255)	nome character varying(255)
1	CPF 11111111	Pessoa Fisica 01
2	CPF 22222222	Pessoa Fisica 02

Figura 3.12: Estrutura da tabela utilizada por herança TABLE_PER_CLASS

```
select * from pessoajuridica
```

ıt pane

ta Output | Explain | Messages | History

id integer	cnpj character varying(255)	nome character varying(255)
3	CNPJ 44444444	Pessoa Juridica
4	CNPJ 55555555	Pessoa Juridica
5	CNPJ 66666666	Pessoa Juridica

Figura 3.13: Estrutura da tabela utilizada por herança TABLE_PER_CLASS

Ao utilizar `TABLE_PER_CLASS`, os atributos presentes nas superclasse estarão também presentes nas tabelas das classes filhas. Por isso, nas tabelas `pessoajuridica` e `pessoafisica` era possível encontrar a coluna `nome`, mas note que esse atributo pertence à superclasse `Pessoa`.

A vantagem dessa abordagem é sua melhor performance na hora de retornar uma entity, afinal todos os dados de uma entity estão em uma tabela apenas.

As desvantagens dessa abordagem são:

- Colunas repetidas — as colunas das superclasses estarão repetidas nas classes filhas. Em nosso caso, a coluna `nome` estará repetida nas tabelas `pessoajuridica` e `pessoafisica`;
- Pode gerar diversos *UNIONs* — se a consulta trouxer todas as *Entities* da herança um número grande de UNION (comando sql) pode ser gerado e atrapalhar a performance do banco de dados.

3.6 Trabalhe com os Embedded Objects

Imagine uma tabela onde temos os dados das pessoas e seus respectivos endereços. Essa é uma abordagem comum encontrada em banco de dados mais antigos, ou para evitar joins entre tabelas para otimizar consultas. Veja a imagem 3.14 da estrutura de uma tabela.

id	nome	idade	end_casa	end_numero	end_bairro
1	jose	44	Rua A	2	Centro
2	maria	23	Rua B	53	Zona Sul
3	joao	64	Rua C	234	Sem Zona
4	feliciana	17	Rua D	77	Principal

Figura 3.14: Tabela com dados que poderiam pertencer a classes distintas

É possível ver nesse exemplo de tabela duas classes distintas que seriam `Pessoa` e `Endereco`. Existem colunas que pertenceriam a `Pessoa` (`nome`, `idade`) e colunas que pertenceriam a `Endereco` (`end_casa`, `end_numero` e `end_bairro`).

Se fôssemos mapear essa entity faríamos como a seguir:

```
@Entity
public class Pessoa{
    @Id
    private int id;
    private String nome;
    private int idade;
    @Column(name = "end_casa")
    private String enderecoCasa;
    @Column(name = "end_numero")
    private int enderecoNumero;
    @Column(name = "end_bairro")
    private String enderecoBairro;

    // get + set e outras coisas
}
```

Veja que a entity `Pessoa` está refletindo atributos que estão mais ligados ao endereço dela do que a ela em si. O ideal seria abstrair essas informações em outra classe para fazer algo como `pessoa.getEndereco()`.

É possível utilizar a entity `Pessoa` e dentro dela mapear um objeto `Endereco` sem que `Endereco` seja uma entity. Veja adiante o código de `Pessoa` e `Endereco`.

```
import javax.persistence.*;

@Entity
```

```
@Table(name = "pessoa")
public class Pessoa {

    @Id
    private int id;
    private String nome;
    private int idade;

    @Embedded
    private Endereco endereco;

    // get and set
}
```

Na entity Pessoa é possível ver o atributo Endereco anotado com @Embedded, que indica à JPA que ela deve entender Endereco como uma classe comum e não uma entity. Todos os campos relacionados a ela serão mapeados para dentro de Pessoa.

```
import javax.persistence.*;

@Embeddable
public class Endereco {

    @Column(name = "end_casa")
    private String rua;

    @Column(name = "end_bairro")
    private String bairro;

    @Column(name = "end_numero")
    private int numero;

    // get and set
}
```

A classe Endereco tem as anotações da JPA @Column que indicam o nome da coluna a ser lida, mas caso o nome do atributo fosse igual ao nome da coluna, não haveria necessidade da anotação. A classe Endereco não é uma entity, mas está anotada com @Embeddable que indica à JPA que essa classe pode ser 'anexada' a uma entity.

Agora, imagine que a entity `Pessoa` tenha endereço residencial e endereço do trabalho. Até então vimos como retirar o endereço residencial — mas seria o correto criar outra classe apenas para adicionar os campos do endereço do trabalho?

Existe a opção de utilizar o mesmo objeto, mas utilizando a anotação `@AttributeOverride` seria possível utilizar o objeto endereço antes configurado para o endereço pessoal. Veja a seguir como é utilizada a anotação `@AttributeOverride`:

```
@Entity
public class PessoaEndereco {
    @Id
    private int id;
    private String nome;
    private int idade;

    @Embedded
    private Endereco enderecoPessoal;

    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name="rua",
                           column=@Column(name="end_trabalho")),
        @AttributeOverride(name="bairro",
                           column=@Column(name="end_bairro_trabalho")),
        @AttributeOverride(name="numero",
                           column=@Column(name="end_numero_trabalho"))
    })
    private Endereco enderecoTrabalho;

    // get and set
```

No código é possível ver a anotação `@AttributeOverrides` que serve para agrupar as anotações `@AttributeOverride`. Na `@AttributeOverride` definimos um atributo `name` que aponta para o nome do atributo que se encontra dentro da classe, e o atributo `column` que é a coluna definida no banco de dados.

Conclusão

Essa funcionalidade serve para organizar melhor a orientação a objetos de sua aplicação.

Outra vantagem dessa abordagem é que podemos ter métodos específicos de endereço dentro desse objeto embedded e não necessariamente dentro da entity Pessoa.

3.7 Mapeie enums e lista de valores

O enum é muito utilizado em nossas aplicações do dia a dia. É possível utilizá-lo com perfil de acesso de um usuário (ADMIN, MANAGER), status de uma compra (APPROVED, DENIED) e muitos outros cenários. Agora, como mapeá-las em sua entidade? Image os enums a seguir:

```
public enum PapelUsuario{
    ADMIN, MANAGER
}

public enum StatusCompra{
    APPROVED, DENIED
}
```

E para utilizar os enums veja o exemplo:

```
@Entity
public class MinhaEntity {

    @Enumerated // OU @Enumerated(EnumType.ORDINAL)
    private PapelUsuario papel;

    @Enumerated(EnumType.STRING)
    private StatusCompra status;
}
```

Note que acima de cada enum é possível encontrar a anotação @Enumerated que indica à JPA como armazenar o valor do enum no banco de dados. O primeiro enum não tem nenhum parâmetro ou pode ser adicionado o valor padrão que é ORDINAL, que quer dizer, o valor salvo no banco de dados será um número. Quando o ORDINAL é utilizado, o valor do índice (ordem em que os atributos do enum foram declarado) será salvo. PapelUsuario.ADMIN será índice 0, e assim por diante. EnumType.STRING ao configurar o enum com essa opção, a JPA salvará o valor nominal do atributo, ou seja, APPROVED, DENIED.

Cada abordagem tem sua vantagem e desvantagem:

- `ORDINAL` — o valor salvo no banco de dados será um valor numérico. Queries executadas nesse campo têm melhor desempenho do que em um campo String (varchar). O problema do `ORDINAL` é que caso o desenvolvedor mude a ordem dos itens do `enum` haverá problema com relação aos dados. Caso a ordem mude de `ADMIN, MANAGER` para `MANAGER, ADMIN` o banco de dados não é atualizado automaticamente e, com isso, quem era `ADMIN` virou gerente e vice-versa. Uma *pequena* desvantagem que se encontra é na hora de investigação e análises feitas diretamente no banco de dados. Um desenvolvedor novato no projeto, após uma consulta em uma tabela, poderá ter dificuldade em descobrir/lembrar que 1 é `ADMIN`, 2 é `MANAGED` e assim por diante.

- `STRING` — o valor salvo no banco de dados será textual. Uma vantagem em salvar o texto e não o índice é que, caso o `enum` mude de ordem, isso não afetará os dados no banco de dados. Outra vantagem é a facilidade em visualizar os dados na tabela, pois na coluna `papel_do_usuario` estaria salvo `ADMIN` e não o número 1 no caso do `ORDINAL`. A desvantagem dessa abordagem seria perda de desempenho que acontece em consulta feita em `String (varchar)`. Esse problema pode ser facilmente contornado criando um índice na coluna. No dia a dia, prefira guardar como texto.

Agora imagine um requisito no qual precisamos que um usuário possa ter diversos e-mails e que ele também tenha mais de um perfil, mapeado através de uma Enum. Para ajudar a resolver esse requisito não é necessário criar uma entity `Email`, basta utilizar a anotação `@ElementCollection`, que fará com que a JPA armazene, automaticamente, esses valores em uma tabela separada.

```
import java.util.*;
import javax.persistence.*;

@Entity
public class Pessoa {

    @Id
    @GeneratedValue
    private int id;

    private String nome;

    @ElementCollection
```

```
    @CollectionTable(name = "pessoa_tem_emails")
    private Set<String> emails;

    @ElementCollection(targetClass = PerfilDoUsuario.class)
    @Enumerated(EnumType.STRING)
    private List<PerfilDoUsuario> perfis;

    // get and set
}
```

É possível ver a presença da anotação `@ElementCollection` sobre o atributo `emails`. Foi utilizada também a anotação `@CollectionTable` que serve para personalizar em qual tabela as listas devem ser salvas. Caso ela não estivesse presente, a JPA procuraria por uma tabela chamada `pessoa_perfis`. Com isso, o padrão utilizado para a definição do nome da tabela é: `nome da classe + _ + nome do atributo`.

A tabela `pessoa_tem_emails` terá como chave estrangeira o ID da `Pessoa`. Veja a imagem a seguir para entender melhor como ficará a tabela:

	pessoa_id integer	email character v
1	1	jose@ema

Figura 3.15: pessoa_tem_emails

Para finalizar, veja o código do `PerfilDoUsuario`. Note que não existe configuração adicional que seja necessária.

```
public enum PerfilDoUsuario {
    ADMINISTRADOR, USUARIO, DONO_DA_EMPRESA
}
```

Conclusão

Utilizar `enum` é fácil, mas é preciso ter muito cuidado quando falamos em utilizar de modo `ORDINAL` ou `STRING`.

A anotação `ElementCollection` é prática e facilita muito quando precisamos armazenar valores simples em uma entity, evitando a criação de uma nova entity.

Capítulo 4

Entenda e mapeie corretamente os relacionamentos

4.1 Use os relacionamentos

Uma das vantagens de uma linguagem orientada a objetos é a realização de relacionamentos. Podemos ter uma classe especializada em algo se relacionando com outra classe que também tem sua especialização. Exemplos de relacionamentos são facilmente encontrados quando construímos uma aplicação: um aluno pode ter diversas matérias e cada matéria pode ter diversos alunos; um aluno pode ter apenas uma nota na matéria e a nota deve pertencer apenas a um aluno; cada professor pode ter várias turmas, mas uma turma só pode ter um professor.

A JPA tem suas anotações e suas regras para o correto funcionamento desses relacionamentos. Basta configurá-las de modo errado que diversos problemas aparecerão, pois pequenos detalhes não foram respeitados. É aí que precisamos ter cuidado.

Vamos ver a seguir como configurar corretamente cada relacionamento.

4.1.1 Relacionamentos com @OneToOne

O relacionamento `@OneToOne` é facilmente entendido com `Pessoa` tem um `Endereco`. Note que esse é um caso bem simples que com Java puro pode ser facilmente retratado como:

```
public class Pessoa {
    private id;
    private nome;

    private Endereco endereco;
    // getters e setters
}

public class Endereco {
    private id;
    private String nomeRua;

    // getters e setters
}
```

O interessante é que a JPA respeita totalmente o relacionamento criado entre `Pessoa` e `Casa`, afinal, esse é o objetivo de uma ferramenta ORM, programamos orientado a objetos e as características são adaptadas para o mundo relacional. Para notificar a JPA que ela deve tratar o relacionamento, basta atualizar as classes para como veremos a seguir:

```
@Entity
public class Pessoa {
    @Id
    private id;
    private nome;

    @OneToOne
    private Endereco endereco;
    // get and set
}

@Entity
```

```
public class Endereco {
    @Id
    private id;
    private String nomeRua;
}
```

Note que agora as classes `Pessoa` e `Casa` foram transformadas em *Entities*. A entity `Casa` continua do mesmo modo quanto aos atributos, mas a entity `Pessoa`, apesar de ter os mesmos atributos, agora tem uma nova anotação, a `@OneToOne`, que informa à JPA que existe um relacionamento entre essas duas entidades que deve ser respeitado.

O mais interessante é como esse relacionamento é refletido no banco de dados. Primeiro, a entidade `Endereco`:

Figura 4.1: Tabela Endereço

Agora, veja a tabela da entidade `Pessoa`:

Figura 4.2: Tabela Pessoa com relacionamento para tabela Endereco

Como nenhuma configuração foi definida para indicar qual o nome da chave estrangeira, a JPA procurará na tabela por uma coluna chamada `casa_id`. O comportamento padrão da JPA é procurar pelo nome da entity + o nome de seu atributo anotado com `@Id`.

O primeiro princípio necessário para a correta utilização dos relacionamentos é: **todo relacionamento tem que ter um lado dominante**. O que quer dizer lado

dominante? Qual tabela do banco de dados terá a chave estrangeira. No exemplo que vimos, apenas a entity `Pessoa` conhece a entity `Casa`. Desse modo é possível afirmar que a chave estrangeira estará na tabela `Pessoa`.

É possível definir o nome da chave estrangeira através da anotação `@JoinColum` como no código:

```
@OneToOne
@JoinColumn(name = "chave_do_endereco")
private Endereco endereco;
```

A anotação tem o atributo *name* para indicar qual deve ser o nome da chave localizada na tabela. No caso anterior haveria uma coluna com o nome *"chave_do_endereco"*.

O segundo princípio a se entender sobre relacionamentos é que **pode existir relacionamento Unidirecional ou Bidirecional**. Um relacionamento unidirecional é o relacionamento em que apenas uma entity conhece a outra; é exatamente o exemplo mostrado aqui entre `Pessoa` e `Endereco`. Apenas a entity `Pessoa` tem referência à entity `Endereco`. É assim que se define um relacionamento unidirecional.

Imagine que em uma tela que lista todos os endereços de funcionários exista um botão para listar o funcionário que mora lá. Seria mais simples fazer `endereco.getPessoa();` do que ter que realizar uma consulta apenas para trazer a pessoa relacionada.

Para transformar um relacionamento em bidirecional será preciso alterar apenas a entity `Endereco`.

```
@Entity
public class Endereco {
    @Id
    private id;
    private String nomeRua;

    @OneToOne
    private Pessoa pessoa;
    // get and set
}
```

Agora temos um relacionamento bidirecional no qual todas as *Entities* envolvidas se conhecem. Esse relacionamento como está configurado funciona? A resposta é sim. Esse relacionamento está corretamente configurado? A resposta é não.

Vamos revisar o primeiro princípio visto sobre relacionamento: **todo relacionamento tem que ter um lado dominante**. No exemplo bidirecional no qual `Pessoa` tem referência a `Endereco` e vice-versa, quem é o lado dominante? Do modo como está o mapeamento a JPA não entenderá que o relacionamento é bidirecional, mas entenderá que cada ponta do relacionamento é um relacionamento único. Dessa forma, `Pessoa pessoa` é um relacionamento e `Endereco endereco` é outro relacionamento `@OneToOne`.

Qual o problema que a JPA terá caso ele não encontre um lado dominante? Ela vai precisar de duas chaves estrangeiras, uma na tabela da entity `Pessoa` e outra na tabela entity `Endereco`. Quando não existe um lado dominante no relacionamento, a JPA analisará cada relacionamento como sendo único. A JPA entenderá que existe um relacionamento único de `Pessoa` para `Endereco`, e também entenderá como único o relacionamento `Endereco` para `Pessoa`. Ou seja, ao invés de um só relacionamento, temos dois. Uma confusão.

Veja na imagem a seguir como ficou a tabela `endereco`:

Figura 4.3: Tabela Endereco com uma chave estrangeira apontando para Pessoa

Note na imagem anterior que como não foi definido o lado dominante agora ambas as tabelas terão uma chave estrangeira. Para determinar o lado dominante é bem simples, faça como a seguir:

```
import javax.persistence.*;

@Entity
public class Pessoa{
    @Id
    private id;
    private nome;

    @OneToOne
    private Endereco endereco;
```

```
    // get and set
}

import javax.persistence.*;
@Entity
public class Endereco{
    @Id
    private id;
    private String nomeRua;

    @OneToOne(mappedBy="endereco")
    private Pessoa pessoa;
    // get and set
}
```

Adicionamos o atributo `mappedBy` no relacionamento que não for dominante. Como nós queremos que `Endereco` seja o lado não dominante, definimos que seu relacionamento com `Pessoa` será marcado com `mappedBy`. Em nosso exemplo, `Pessoa` tem um atributo chamado `endereco`, e é esse o nome que deve estar no atributo `mappedBy`.

Por fim, há um terceiro conceito importante que precisamos ter em mente: **a JPA não fará o relacionamento entre classes automaticamente**.

Como assim relacionando os dois lados corretamente? Se usado de modo errado, um relacionamento bidirecional, pode causar mais dano do que benefício. Veja o código a seguir:

```
entityManager.getTransaction().begin();
pessoa.setCasa(casa);
entityManager.getTransaction().commit();
```

Levando em conta que o nosso relacionamento é bidirecional e apenas o comando `pessoa.setEndereco(endereco);` foi executado, esse relacionamento apresentará algum tipo de comportamento *"estranho"*. A JPA trabalha com o conceito básico do Java de referência: se uma referência não foi passada para o outro objeto, esse relacionamento não existirá do outro lado, e em nosso caso, `casa` não existiria para `pessoa`. Em um relacionamento bidirecional, fazer apenas `pessoa.setCasa(casa)` não é suficiente, mas seria necessário fazer `casa.setPessoa(pessoa);`.

A JPA não fará o trabalho de trocar as referências em um relacionamento bidirecional, é necessário que o desenvolvedor saiba e faça isso.
–

4.1.2 Cuidados com o @OneToMany e @ManyToOne

Imagine um sistema de Petshop no qual uma pessoa pode ter vários cachorros e um cachorro pode ter um dono. Nesse caso não estamos falando mais de um relacionamento um para um, pois a pessoa pode ter de 0 a vários cachorros. O melhor relacionamento para representar esse tipo situação seria o relacionamento OneToMany.

Vamos começar analisando a entidade `Pessoa`:

```
import javax.persistence.*;

@Entity
public class Pessoa{
    // outras informações
    @OneToMany
    private List<Cachorro> cachorros;
}
```

Veja que a entidade `Pessoa` agora tem uma lista de `Cachorro`, mas com a anotação `@OneToMany` acima dela. Essa anotação indica à JPA que um relacionamento existe e ele deve gerenciar.

Veja como ficará nosso banco de dados:

Figura 4.4: Tabela pessoa

Figura 4.5: Tabela pessoa_cachorro

Figura 4.6: Tabela cachorro

Do modo como anotamos nosso relacionamento, a JPA precisará de uma tabela extra para armazenar os valores. Por isso temos a tabela `pessoa`, a `cachorro` e a tabela `pessoa_cachorro` para realizar o relacionamento entre elas.

Para a JPA 2 é possível ter o relacionamento unidirecional, no qual apenas `Pessoa` tem referência para `Cachorro`, sem a tabela adicional. Para não precisar de uma tabela adicional vamos utilizar a anotação `@JoinColumn`:

```
import javax.persistence.*;

@Entity
public class Pessoa{
    // outras informações
    @OneToMany
    @JoinColumn(name = "pessoa_id")
    private List<Cachorro> cachorros;
}
```

Uma vez que utilizamos a anotação `@JoinColumn`, a JPA procurará por uma coluna chamada `pessoa_id` dentro da tabela `cachorro`. Veja como ficará a tabela `cachorro`:

Figura 4.7: Tabela cachorro com chave estrangeira para pessoa

Como `Cachorro` só pode ter uma `Pessoa` a chave fica na tabela `Cachorro` e não será mais necessário ter uma tabela só para o relacionamento.

Imagine agora que estamos a listar `cachorros` e queremos exibir quem é o dono de cada cachorro. É nesse momento que nosso relacionamento deve virar bidirecional. Para que o relacionamento seja bidirecional precisamos alterar a entidade `Cachorro`:

```
@Entity
public class Cachorro {
    @Id
    @GeneratedValue
    private int id;

    private String nome;

    @ManyToOne
    @JoinColumn(name = "pessoa_id")
    private Pessoa pessoa;
}
```

Veja que agora a entidade `Cachorro` tem uma referência para `Pessoa` e com a anotação `@JoinColumn`. E apenas um pequeno ajuste precisa ser feito na entidade `Pessoa`:

```
@Entity
public class Pessoa {
    // outras coisas
    @OneToMany(mappedBy = "pessoa")
    private List<Cachorro> cachorros;
}
```

O atributo `mappedBy` foi utilizado para definir que a entidade `Cachorro` é a dona do relacionamento, é a tabela `cachorro` que terá a chave estrangeira.

E novamente é preciso reforçar que será sempre necessário fazer o relacionamento dos dois lados. Para que o código funcione corretamente deve-se fazer:

```
cachorro.setPessoa(pessoa);
pessoa.getCachorros().add(cachorro);
```

4.1.3 @ManyToMany

Imagine agora um sistema no qual será registrado todos os trabalhos por onde uma pessoa já passou. Vamos ver como ficaria um relacionamento `@ManyToMany`. Esse relacionamento pode ser aplicado no seguinte caso: uma `Pessoa` tem diversos `trabalhos` e um `Trabalho` tem diversas `pessoas`. No caso de um relacionamento `@ManyToMany` é necessário uma tabela que faça a união das tabelas.

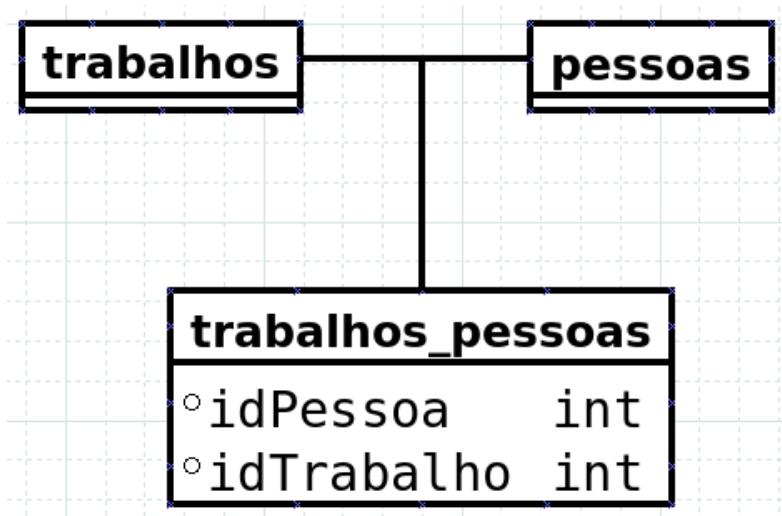



Figura 4.8: Tabela extra para um relacionamento @ManyToMany

É possível ver na imagem 4.8 que uma tabela é utilizada para fazer o relacionamento entre `pessoas` e `trabalhos`. Nessa abordagem não existe chave estrangeira nas tabelas das *Entities*.

Vamos começar mapeando a entity `Pessoa`, deixando-a como o lado dominante do relacionamento.

```
@Entity
public class Pessoa{
    // outras informações
    @ManyToMany
    @JoinTable(name = "trabalhos_pessoas")
    private List<Trabalho> trabalhos;
}
```

A entity `Pessoa` está sendo marcada como dona do relacionamento pela anotação `JoinTable`, e aponta o nome da tabela de relacionamento. Do modo como está mapeado anteriormente, o relacionamento é unidirecional com uma tabela extra que armazenará as chaves de cada tabela.

E se no caso precisássemos criar o relacionamento do lado do `Trabalho` também? Para definir qual o lado não dominante em um relacionamento bidirecional basta fazer como a seguir:

```
@Entity
public class Trabalho {

    @Id
    @GeneratedValue
    private int id;

    private String nome;

    @ManyToMany(mappedBy = "trabalhos")
    private List<Pessoa> funcionarios;

    // get and set
}
```

Veja que para utilizar um relacionamento bidirecional, a lista `funcionarios` foi mapeada utilizando o `mappedBy` apontando para `trabalhos`. O que aconteceria caso não declarássemos lado dominante? A JPA procuraria por uma tabela de relacionamento para a ponta `funcionarios` e uma tabela para a ponta `trabalhos`.

Um relacionamento com problemas teria as tabelas como na imagem adiante:

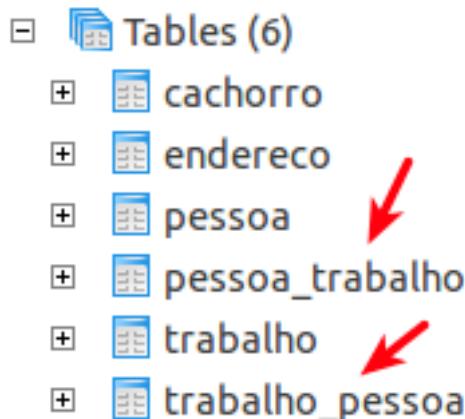



Figura 4.9: Problema de não definir um lado dominante

Assim como já dito anteriormente é necessário sempre realizar o relacionamento dos dois lados da entidade:

```
pessoa.getTrabalhos().add(trabalho);
trabalho.getPessoas().add(pessoa);
```

Conclusão

Relacionamentos sempre existirão nos projetos Java, mas é preciso bastante cuidado ao utilizá-los. Um relacionamento mapeado de modo errado pode levar a dados inconsistentes e acarretar uma perda de credibilidade do projeto.

Quando os relacionamentos são configurados corretamente, sua aplicação não terá problema de dados e relacionamentos.

4.2 Entenda como funciona o Cascade

Em diversas situações, é comum ter duas ou mais entidades envolvidas na transação. Imagine uma tela onde temos como entrada de dados o nome, idade, endereço, CEP. Podemos ter uma entidade `Endereco` e outra `Pessoa`.

Para a gravação das informações, poderíamos ter:

```
entityManager.getTransaction().begin();
Endereco endereco = new Endereco();
```

```
endereco.setRua("30 de Fevereiro");
endereco.setNumero("43A");
endereco.setComplemento("Edifício Itaoca da Pedra");

Pessoa pessoa = entityManager.find(Pessoa.class);
pessoa.setEndereco(endereco);

entityManager.getTransaction().commit();
entityManager.close();
```

Esse código parece perfeitamente normal, mas ele geraria uma `org.hibernate.TransientObjectException` caso esteja usando o Hibernate. Para o EclipseLink, apareceria a seguinte mensagem:

```
Caused by: java.lang.IllegalStateException: During synchronization
a new object was found through a relationship that was not
marked cascade PERSIST
```

Para que possamos realmente saber o que aconteceu, é preciso entender um ponto importantíssimo da JPA, que são os estados em que uma entidade pode estar.

Uma das grandes vantagens da JPA é o rastreamento que ele é capaz de fazer em cada entity envolvida na transação, cada entity quer for *'anexada'* na transação poderá ser criada, alterada e até mesmo removida do banco de dados.

Para que a JPA possa exercer qualquer alteração automaticamente na entidade, ele precisa saber de onde ela vem e o que acontecerá com ela até o final da transação. Ele consegue ter esse controle através dos estados de uma entity, que podem ser *managed*, *removed* ou *detached*. O foco desse capítulo será nos estados *managed* e *detached*.

O estado *managed* acontece quando a entidade está sincronizada à transação. Uma das formas de conseguirmos isso é através do método `find`, que nos devolve uma entidade a partir do seu identificador. O objeto devolvido estará no estado *managed*.

```
entityManager.getTransaction().begin();
Pessoa pessoa = entityManager.find(Pessoa.class, 33);
pessoa.setNome("Jose de Arimatéia");
entityManager.getTransaction().commit();
```

A entidade `pessoa` foi recuperada do banco de dados e teve o atributo `nome` alterado. O interessante desse código é que ao realizar o `commit()` essa alteração será refletida no banco de dados.

Mas o que há de tão especial nisso? Repare que não foi preciso executar `entityManager.merge()` para que a alteração seja refletida no banco de dados. Todas as alterações realizadas em uma entity que está *managed* serão automaticamente refletidas no banco de dados.

Outro conceito interessante é que a entity, após ser *managed*, é colocada dentro de uma estrutura chamada *Persistence Context*, que é criada junto da `EntityManager`.

É possível entender o *Persistence Context* como uma sacola, e nessa sacola serão colocadas todas as entities que passarem pela transação. É justamente desse modo que a JPA consegue rastrear o que acontece com cada entity. Mas fique atento com consultas ao banco de dados que vão trazer muitos resultados, pois essa sacola poderá ficar muito grande e estourar a memória do servidor. A solução para esse problema é paginar a consulta, uma técnica que veremos no capítulo 8.2.

Veja as imagens 4.10 e 4.11 para exemplificar melhor quando uma entity está *managed*.

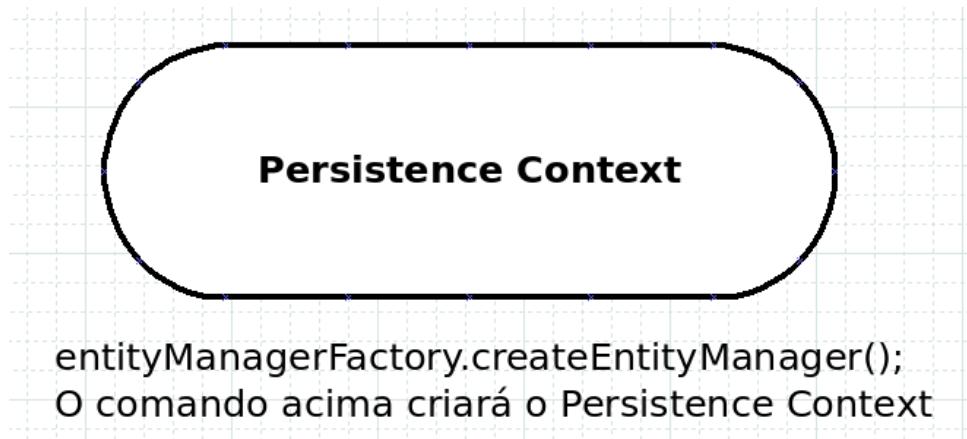



Figura 4.10: Criando Persistence Context

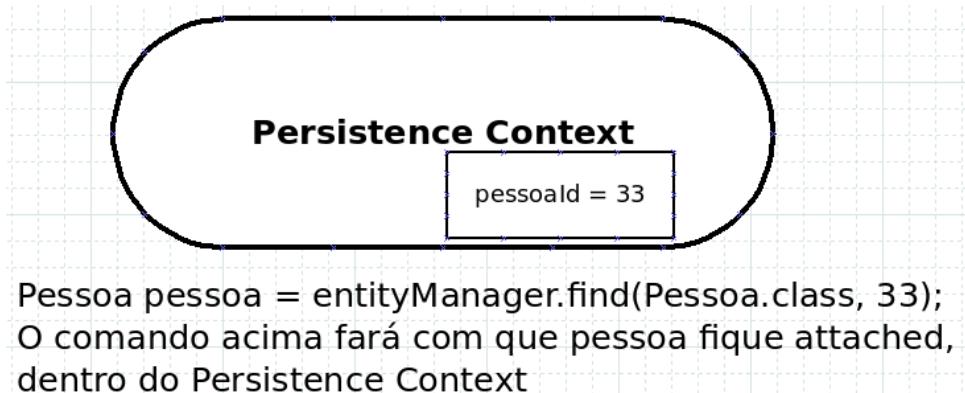



Figura 4.11: Adicionando Entity ao Persistence Context

É possível dizer que para uma entity estar *managed*, ela passará pelo *EntityManager*, seja por comandos diretamente executados pelo *EntityManager* (`find`, `persist`, `merge`, `getReference` etc.), seja por queries (`entityManager.createQuery()`). Ao utilizar o *EntityManager* para trazer qualquer entity do banco de dados, ela estará *managed*.

Mas e se estivermos não estivermos buscando uma pessoa do banco de dados, e sim criando uma entity nova e que não ainda não possui ligação nenhuma com a `EntityManager`? No exemplo a seguir, repare no objeto `endereco`, que é relacionado a `pessoa`.

```
entityManager.getTransaction().begin();

Endereco endereco = new Endereco();
endereco.setId(33);

Pessoa pessoa = entityManager.find(Pessoa.class);
pessoa.setEndereco(endereco);

entityManager.getTransaction().commit();
```

Nesse caso, o `id` do endereço está sendo passado explicitamente. No entanto, em uma aplicação web, é comum recebermos o objeto populado com o `id`, por exemplo, através da seleção em um combo.

A exceção acontecerá ao final da transação, visto que relacionamos uma entity *managed* a uma entity *detached*. Quando a JPA fizer o commit da transação, ela

analisará os objetos relacionados para salvar as informações necessárias no banco de dados.

detached é o termo utilizado para a seguinte situação: uma entity passou pela transação e a transação foi finalizada. Imagine uma tela onde o usuário executa uma pesquisa de pessoa pelo id 33:

```
entityManager.getTransaction().begin();
Pessoa pessoa = entityManager.find(33, Pessoa.class);
entityManager.getTransaction().commit();
entityManager.close();
// nessa linha a entidade pessoa está detached
```

Uma vez que a entidade `pessoa` esteja *detached*, a JPA não monitorará mais o que acontecer com ela.

Note que `endereco` em nenhum momento passou pelo *EntityManager*, ou seja, a JPA não tem a mínima ideia de quem é essa entity para o *Persistence Context* atual.

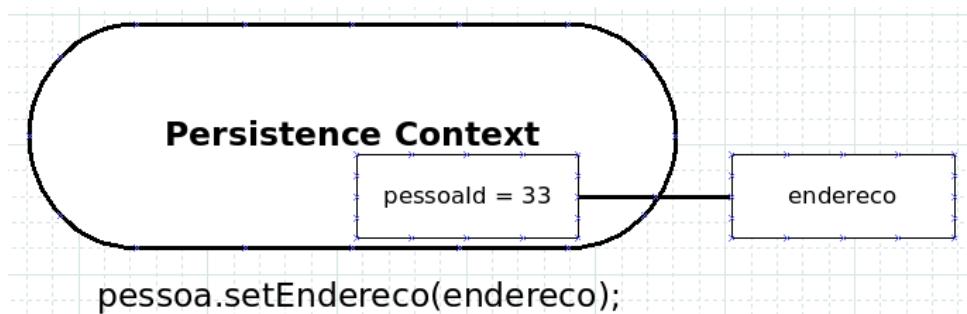



Figura 4.12: Adicionando Entity ao Persistence Context

A imagem 4.12 mostra por que o erro acontece ao criar um relacionamento entre uma entity *managed* e uma entity *detached*. A JPA não consegue saber de onde veio a entity que está fora do *Persistence Context* atual. Mesmo que `endereco` exista no banco de dados, como ela está fora da transação, a JPA não consegue reconhecê-la. Uma entity fora do *Persistence Context* é chamada de *detached*.

Esta situação de encontrar uma *Entity detached* pode acontecer com qualquer ação a ser executada no banco de dados como: INSERT, UPDATE e DELETE.

Para ajudar o desenvolvedor com as *Entities* que estão *detached*, a JPA implantou o conceito de *Cascade*, de certa forma similar ao que alguns bancos de dados supor-

tam. Simplesmente quer dizer que a ação (`persist`, `merge` ou `remove`) que for disparada em uma entity deve ser refletida para as demais *Entities* relacionadas.

O *Cascade* é muito útil quando queremos que a JPA execute ações em seus relacionamento. Imagine que temos o relacionamento `Pessoa` e `Endereco`; quando uma `Pessoa` for salva no banco de dados, queremos que esse endereço já seja salvo também. Veja o código a seguir:

```
entityManager.getTransaction().begin();
Pessoa pessoa = new Pessoa();
populaPessoaComDados(pessoa);
Endereco endereco = new Endereco();
populaEnderecoComDados(endereco);
pessoa.setEndereco(endereco);
entityManager.persist(pessoa);
entityManager.getTransaction().commit();
```

É possível ver no código anterior que apenas `pessoa` foi persistido, mas uma vez que utilizamos o `Cascade`, ao persistir `pessoa`, queríamos que `endereco` fosse automaticamente salvo. Podemos dizer que o estado do endereço é um estado novo `new`; ele ainda não foi persistido e com a ajuda da JPA isso já será feito. Poderíamos também querer uma ação de `Cascade` na hora da exclusão: `entityManager.remove(pessoa)`. Desse modo, ao excluir `pessoa` o `endereco` seria excluído também do banco de dados caso o `Cascade` fosse corretamente configurado.

As definições do *Cascade* são passadas dentro das anotações `@OneToOne`, `@OneToMany` e `@ManyToMany`, e as possibilidades de valores para o *Cascade* podem ser encontradas dentro do *enum* `javax.persistence.CascadeType`.

Vamos ver a seguir um detalhamento do de cada tipo de *Cascade*:

- `CascadeType.PERSIST` – disparado toda vez que uma nova entity for inserida no banco de dados pelo comando: `entityManager.persist()`;
- `CascadeType.DETACH` – disparado toda vez que uma entity é retirada do `Persistence Context`. Comandos que podem disparar essa ação: `entityManager.detach()`, `entityManager.clear()`. Ocorrerá um *detach* também quando o `Persistence Context` deixar de existir;
- `CascadeType.MERGE` – disparado toda vez que uma alteração é executada em uma entity. Essa alteração pode acontecer ao final de uma tran-

sação com a qual uma *managed Entity* foi alterada (4.2), ou pelo comando `entityManager.merge();`

- `CascadeType.REFRESH` – disparada quando uma entity for atualizada com informações no banco de dados. Comando: `entityManager.refresh();`
- `CascadeType.REMOVE` – disparado quando uma entity é removida (apagada) do banco de dados, os relacionamentos marcados também serão eliminados. O comando utilizado é o `entityManager.remove();`
- `CascadeType.ALL` – todos os eventos anteriores serão sempre refletidos nas *Entities* relacionadas.

A função do *Cascade* é propagar a ação que acabou de ser executada. Para configurá-lo, basta indicar no relacionamento:

```
@Entity
public class Pessoa{
    // outros métodos omitidos

    @OneToOne(cascade = CascadeType.PERSIST)
    private Endereco endereco;
}
```

Ao aplicar o *Cascade*, a JPA replicará toda ação recebida na entity para o relacionamento configurado. A entity `Pessoa` teve seu relacionamento com `Endereco` configurado com `@OneToOne(cascade = CascadeType.PERSIST)`. Com isso, toda vez que `Pessoa` for persistida, o mesmo comando será repassado ao `endereco`, ou seja, a JPA também fará `entityManager.persist(endereco)`.

Dessa forma, o código a seguir, que cria um novo endereço e o associa a uma pessoa, funcionaria sem problemas:

```
entityManager.getTransaction().begin();

Endereco endereco = new Endereco();
endereco.setNome("Rua 33 numero 33");

Pessoa pessoa = new Pessoa();
pessoa.setNome("Gertrudes");
```

```
pessoa.setEndereco(endereco);

entityManager.persist(pessoa);

entityManager.getTransaction().commit();
```

No código é possível ver que em nenhum momento o `endereco` recebeu o comando `persist`. Como o relacionamento entre `pessoa` e `endereco` está marcado com `CascadeType.PERSIST`, a JPA automaticamente vai gravar do `endereco`.

O uso do *Cascade* pode parecer uma arma muito poderosa a princípio, no entanto, é preciso ter muito cuidado ao configurá-lo. Não o use sem ter certeza se realmente é essa ação é desejada. Caso o `CascadeType.REMOVE` ou `CascadeType.ALL` esteja configurado, ao excluir uma entity, o seu relacionamento também será removido e, com isso, você pode perder informações sensíveis à sua aplicação. Imagine se houvesse um *Cascade* de `Endereco` para `Pessoa`, ao excluir `Endereco`, a `Pessoa` também seria excluída. Dependendo da aplicação, isso poderia ser um grande problema.

Além disso, `CascadeType.ALL` pode deixar a aplicação com alguma lentidão caso uma entity tenha muitas listas, e elas sejam marcadas com essa opção. Ao executar um `entityManager.merge()` todas as listas marcadas com o *Cascade* receberiam essa ação também, demandando uma operação em todos os objetos associados.

Por fim, tenha em mente que para disparar o *Cascade* é sempre necessário que a ação seja executada na entity em que ele foi configurado. Veja o código:

```
public class Endereco{
    @OneToOne(mappedBy = "endereco")
    private Pessoa pessoa;
}

public class Pessoa{
    @OneToOne(cascade = CascadeType.PERSIST)
    private Endereco endereco;
}
```

Note que apenas na entity `Pessoa` é encontrado o `cascade = CascadeType.PERSIST`. Se o desenvolvedor fizer o comando `entityManager.persist(endereco)`, o *Cascade* não será disparado. Em

nosso caso, o *Cascade* só funcionará caso o comando seja executado na entity `Pessoa`, como em `entityManager.persist(pessoa)`.

Para finalizar esse assunto, o *Cascade* só será ativado para comandos que passem pelo *EntityManager*. Se um comando do tipo `delete p from Pessoa p` for executado, o *Cascade* não será executado pela JPA. Veremos isso com mais detalhes no capítulo sobre operações em lote (8.3).

Conclusão

A funcionalidade de *Cascade* é muito boa e útil, mas deve ser utilizada com cautela. É comum encontrar pessoas em fóruns que, ao perguntar sobre problemas, mostram suas entidades todas anotadas com `CascadeType.ALL` sem ao menos entender o que isso significa ou faz.

4.3 Entenda como funciona o OrphanRemoval

Imagine um sistema no qual temos um menu totalmente dinâmico, em que o usuário poderia criar itens e subitens. Quando quisesse, ele poderia simplesmente apagar um determinado menu do banco de dados. Imagine que o usuário criou um menu + um subitem como “Relatórios > Meu Relatório”:

Figura 4.13: Menu criado pelo usuário

Após algum tempo o usuário resolve apagar o menu “Relatórios”. Caso a rotina de exclusão não seja corretamente escrita, o subitem “Meu Relatório” poderia ficar “perdido” no banco de dados, ou seja, sem estar relacionada a qualquer menu. Note que a existência do subitem se dá diretamente à existência de um menu; quando temos um subitem sem menu, chamamos esse objeto de órfão, sem o objeto pai que lhe deu origem.

Tecnicamente falando, quando temos um objeto que só possa existir na presença de outro estamos utilizando Composição.

É justamente para evitar problemas de registros órfãos que devemos utilizar o atributo `orphanRemoval` que encontramos nas anotações de relacionamentos.

A função *Orphan Removal* deve ser utilizada em casos de Composição, ou seja, onde uma Entity só pode existir caso outra exista. Sogra só existe se tiver a namorada, endereço só existe caso exista um usuário, mas vamos usar um caso de uso mais simples. Para se ter um `SubItem` é necessário ter um `Menu`, ou seja, sem um `Menu` é impossível o `SubItem` existir. A imagem 4.14 detalha mais esse relacionamento.

Figura 4.14: Relacionamento onde uma Entity depende de outra para existir

É possível perceber no diagrama que a existência do `SubItem` está diretamente ligada a um `Menu`, mas note que um `Menu` pode existir perfeitamente sem `SubItem`. É possível informar à JPA que, quando uma entity `SubItem` não estiver relacionada com nenhum `Menu`, ela deve ser removida (*'delete'*) do banco de dados.

Vamos mapear as *Entities* `Menu` e `SubItem`:

```
@Entity
public class Menu{
    @OneToOne(orphanRemoval=true)
    private SubItem subItem;
    // vamos desconsiderar o resto
}

@Entity
public class SubItem{
    private String nome;
    // vamos desconsiderar o resto
}
```

Veja que a anotação `OneToOne` tem o atributo marcado `orphanRemoval` definido como `true`. E qual a vantagem dessa abordagem? Imagine que o `subItem`

foi removido do `menu`. Basta fazer o código a seguir:

```
entityManager.getTransaction().begin();
Menu menu = entityManager.find(Menu.class, 33);
menu.setSubItem(null);
entityManager.getTransaction().commit();
```

Existe uma condição com a qual o código anterior não funcionará. Caso a entity `SubItem` tivesse relacionamento com qualquer outra entity. Imagine que a entity `SubItem` fosse como a seguir:

```
@Entity
public class SubItem{
    private String nome;

    @OneToMany
    private List<Estilo> estilos;
    // vamos desconsiderar o resto
}
```

É possível ver no novo código da entity `SubItem` que existe um relacionamento com uma lista de `Estilo`. Caso exista itens nessa lista, o relacionamento não será desfeito. Uma entity que tenha qualquer relacionamento com outra entity não será removida pela JPA apenas com a opção `orphanRemoval=true`. É necessário eliminar todo o relacionamento para que a JPA faça a remoção desse objeto *'órfão'*.

Conclusão

Muitos desenvolvedores simplesmente adicionam a opção `orphanRemoval=true` em seus relacionamentos sem se tocar da consequência que isso pode acarretar. No exemplo anterior, imagine que um `SubItem` deve permanecer na base após a exclusão de um `Menu`. Um desenvolvedor pode acabar adicionando o `orphanRemoval` porque viu em algum lugar da internet essa configuração. O que acarretaria a configuração `orphanRemoval` a mais? No caso de um `SubItem` não ter que ser excluído, essa regra seria quebrada e o `SubItem` seria apagado.

`OrphanRemoval` quando corretamente configurado e utilizado é uma ferramenta poderosa e que poupa trabalho manual do desenvolvedor.

4.4 Como utilizar Lazy e Eager Loading corretamente

Quando a JPA executa uma consulta no banco de dados, ela não tem como saber se aquela classe tem ou não muitas informações no banco de dados. Veja a entity a seguir:

```
@Entity
public class Pessoa{
    @Id
    private int id;

    @Lob
    private byte[] fotoPerfil;

    // outras coisas
}
```

Imagine que os usuários estão colocando em seu perfil fotos de até 150MB. Ao buscar uma pessoa pelo seu `id` através do método `find`, a JPA trará o atributo `fotoPerfil`.

Considere que nesse banco de dados haja mais de 1000 usuários. Ao fazer um comando do tipo `select p from Pessoa p`, pode haver um estouro de memória, pois seriam carregados 1.000 objetos, sendo que cada um teria uma foto de aproximadamente 150MB.

Esse comportamento de trazer os atributos no ato da consulta é chamado de `EAGER`. Todo atributo *simples*, ou seja, o que não é relacionamento, será carregado automaticamente. Dessa forma, `byte[] fotoPerfil` seria buscados automaticamente. Mas como faço para não carregá-lo automaticamente e evitar problemas de consumo de memória e performance?

```
@Entity
public class Pessoa{
    @Id
    private int id;

    @Lob
    @Basic(fetch = FetchType.LAZY)
    private byte[] fotoPerfil;
```

```
    // outras coisas
}
```

A anotação `@Basic(fetch = FetchType.LAZY)` está definindo que o campo não deve ser carregado quando aquela entity for recuperada. Agora que o atributo `byte[] fotoPerfil` foi definido como `LAZY`, a JPA não o trará nas consultas futuras.

Ao chamar o método `find`, o atributo `fotoPerfil` só será trazido do banco de dados caso o método `pessoa.getFotoPerfil()` seja executado. Ao buscar por um atributo definido como `LAZY`, uma nova consulta será realizada no banco de dados buscando essa informação.

Mas fique atento, esse tipo de comportamento muda quando falamos de relacionamentos. Nesse caso, temos os seguintes comportamento:

- Toda vez que um relacionamento terminar em `One` (`OneToOne`, `ManyToOne`) ele será por default `EAGER`;
- Toda vez que um relacionamento terminar em `Many` (`OneToMany`, `ManyToMany`) ele será por default `LAZY`.

Uma boa maneira de se lembrar disso é, nos relacionamentos, caso precise trazer muitos objetos, a JPA irá sempre esperar, ou seja, será `LAZY`. Caso seja um só objeto, ele consumirá pouco recurso, e vai ser buscado direto, assim sendo `EAGER`.

Veja a entity:

```
@Entity
public class Pessoa{
    @Id
    private int id;

    @OneToOne
    private Pessoa conjuge;

    @OneToMany
    private List<Pessoa> filhos;
    // outras coisas
}
```

Nesse código temos um relacionamento `EAGER`, `conjuge` e um relacionamento `LAZY` em `filhos`.

Caso não queiramos o comportamento padrão, podemos mudar através do atributo `fetch` da anotação de relacionamento. É preciso ter bastante cuidado ao alterar o comportamento padrão de cada atributo de carregamento. Considere a entidade a seguir:

```
public class Pessoa{
    @OneToMany(fetch = FetchType.EAGER)
    private List<Pessoa> emails;
    // outras coisas
}
```

Imagine que uma `Pessoa` tenha muitos e-mails. Ao carregar uma lista de 1.000 `pessoa` e na entity o relacionamento `List<Email> emails` estaria marcado como `EAGER`. Desse modo ao carregar cada `Pessoa` do banco de dados todos seus `emails` já seriam carregados também. Esse tipo de ação poderia fazer com que aconteça um `OutOfMemoryError`, já que muitos objetos estão sendo carregados para a memória.

4.5 Entenda a LazyInitializationException

Como vimos, é possível que um atributo seja carregado de modo `EAGER` ou `LAZY`. Vamos utilizar a entity a seguir como exemplo:

```
@Entity
public class Pessoa {

    @OneToMany(fetch = FetchType.LAZY)
    private List<Email> emails;
    // outras coisas
}
```

Note que a lista de `emails` está `LAZY`. Dessa forma, para exibir seus valores, a seguinte consulta seria realizada:

```
entityManager.getTransaction().begin();
Pessoa pessoa = entityManager.find(Pessoa.class, 33);
entityManager.getTransaction().commit();
entityManager.close();
return pessoa;
```

Poderíamos exibir essa lista na tela usando JSF, com o seguinte código:

```
<h:dataTable var="poste"
    value="#{pessoasMB.pessoa.emails}">
    <h:column>
        <f:facet name="header">
            Titulo do Email
        </f:facet>
        #{email.titulo}
    </h:column>
</h:dataTable>
```

É um código simples, que irá exibir o nome de cada e-mail enviado por uma `pessoa`. O problema é que ao ser executado, a seguinte mensagem aparecerá:

```
javax.enterprise.resource.webcontainer.jsf.application
(http-127.0.0.1-8080-2)

Error Rendering View 'listarEmails.xhtml':
org.hibernate.LazyInitializationException: failed to lazily initialize
a collection of role: com.model.Pessoa.emails, no session or session
was closed
```

Como vimos, todo relacionamento definido como `LAZY` só será carregado se for acessado. Nesse exemplo, bastaria fazer `pessoa.getEmails()` para que a JPA disparasse uma consulta no banco de dados para trazer os dados da lista. O problema é que quando essa consulta automática é realizada e a conexão com o banco de dados foi finalizada.

As mensagens para esse erro podem variar de acordo com a implementação usada. No caso, foi mostrada o que acontece quando o Hibernate é usado.

Existem diversas maneiras de solucionar esse problema e evitar a `LazyInitializationException`. Vamos aprender as mais eficazes.

4.5.1 Utilizando o método size das listas

A exceção ocorre, pois a lista é buscada tarde demais, quando a conexão já está fechada. Uma solução simples é forçar o carregamento da lista antes disso. Uma forma de fazer isso é invocando o método `size` da lista:

```
entityManager.getTransaction().begin();
Pessoa pessoa = entityManager.find(Pessoa.class, 33);
```

```
pessoa.getEmails().size();

entityManager.getTransaction().commit();
entityManager.close();
return cachorro;
```

Repare na chamada ao método `size`. Desse modo, não haveria problema se, após o término da transação, a lista fosse acessada.

Suas vantagens são:

- Mais fácil de resolver o problema;
- Evita o problema n+1, que veremos no capítulo 4.8.

E suas desvantagens:

- É preciso lembrar que ao final de cada consulta esse comando sempre deve ser executado;
- A linha com o `.size()` poderia facilmente ser removido por engano por algum desenvolvedor.

Esse modo de resolver o problema, é por muitos, considerado uma péssima prática de programação.

4.5.2 Carregamento por anotação

Lembrar de executar um comando para buscar todas as informações do relacionamento pode ser arriscado, já que é muito fácil de ser esquecido. Logo, uma alternativa bem natural é tentar fazer com que tudo seja automático. É justamente isso que podemos fazer indicando `EAGER` no atributo `fetch` da anotação de relacionamento.

```
@Entity
public class Pessoa{

    @OneToMany(fetch = FetchType.EAGER)
    private List<Email> emails;
    // outras coisas
}
```

Ao marcar uma coleção com `EAGER`, toda vez que a entidade for buscada no banco de dados, os dados do relacionamento também virão. Note que apenas os relacionamentos que terminam com `ToMany` são `LAZY` por padrão, ou seja, é nesse caso que você deverá estar atento à essa configuração.

As vantagens dessa abordagem são:

- Fácil de configurar;
- Toda vez que qualquer consulta for realizada na entity a lista será carregada.

Mas é preciso se atentar a uma grande desvantagem:

- Caso uma entity tenha uma lista muito grande ou diversas listas marcadas como `EAGER`, a performance do servidor pode ser impactada, já que muitos objetos serão carregadas e possivelmente de forma desnecessária.

Essa abordagem é boa quando é possível ter a certeza de que a lista terá sempre poucos valores. O impacto no banco de dados seria mínimo ao trazer uma coleção com 3 itens, por exemplo.

[subsection Carregar por OpenSessionInView]

Para as soluções vistas anteriormente temos a desvantagem de a transação ficar aberta pouco tempo, em geral, apenas o tempo de uma consulta. Existe o padrão *OpenSessionInView* que nos ajuda e muito para tratar transação e o problema que está sendo visto neste capítulo.

OpenSessionInView — *OSIV* (ou *TransactionInView*) é um pattern de desenvolvimento muito utilizado no mundo web. A transação ficará aberta enquanto durar o `HttpRequest` do usuário.

Para configurar o *OSIV* é necessário criar um `Filter` no projeto:

```
@WebFilter(urlPatterns={"/*"})
public class JpaControllerFilter implements Filter {
    private EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("Project PU");

    @Override
    public void init(FilterConfig filterConfig)
            throws ServletException {
```

```
    }

    @Override
    public void doFilter(ServletRequest request,
                         ServletResponse response,
                         FilterChain filterChain)
            throws IOException, ServletException {
        EntityManager entityManager = emf.createEntityManager();
        try{
            entityManager.getTransaction().begin();

            // realiza as ações do sistema
            filterChain.doFilter(request, response);

            entityManager.getTransaction().commit();
        } catch (Exception ex){
            if(entityManager != null &&
               entityManager.getTransaction().isActive()){
                entityManager.getTransaction().rollback();
            }
        } finally {
            if(entityManager != null && entityManager.isOpen()){
                entityManager.close();
            }
        }
    }

    @Override
    public void destroy() {

    }
}
```

No exemplo, utilizamos `EntityManagerFactory` criar um `EntityManager`. Note que toda a transação acontece dentro de um *try/catch/finally*; a transação é aberta, depois o comando `filterChain.doFilter(request, response);` chamará os métodos das classes do sistema. Quando toda a ação acabar com sucesso, a transação receberá *commit* e o `entityManager` finalizado no `finally`. Note que, caso aconteça uma exception, o rollback da transação será realizado dentro do `Catch`.

As vantagens dessa abordagem são:

- Não existe a necessidade de alteração na entity;
- Todas as classes se beneficiarão da mudança.

As desvantagens dessa abordagem são:

- O código anterior não é aplicável ao JSE; seria necessário aplicar o mesmo conceito em uma classe que controlaria todo o fluxo do projeto;
- A transação deve ser controlada manualmente pelo usuário no filtro, diferentemente de um servidor JEE que faz todo o trabalho;
- Deve ter bastante cuidado ao tratar *Exceptions* para reverter transações adequadamente.

[/subsection]

4.5.3 Stateful EJB

Para os desenvolvedores que utilizam EJB, uma boa alternativa é utilizar um EJB do tipo `Stateful` aliado ao `PersistenceContext Extended`. Veja o código a seguir:

```
import javax.ejb.Stateful;
import javax.persistence.*;

@Stateful
public class CachorroStateful {
    @PersistenceContext(unitName = "LazyPU",
            type=PersistenceContextType.EXTENDED)
    private EntityManager entityManager;

    public Cachorro find(Integer id) {
        Cachorro cachorro = entityManager.find(Cachorro.class, id);

        return cachorro;
    }
}
```

Ao definir um `EntityManager` como `PersistenceContextType.EXTENDED`, o EJB sempre executará a busca no banco de dados.

As vantagens dessa abordagem são:

- O container do EJB controlará a transação;
- Não necessita alterar o modelo.

As desvantagens dessa abordagem são:

- Não é aplicável ao Java SE;
- Ocupará memória enquanto não for 'destruído', seja por timeout ou quando a referência ao EJB não for mais utilizada.

Essa solução se aplica a ambiente Java EE apenas.

4.5.4 Carregando por Query com Join

É possível também indicar à JPA qual relacionamento carregar de modo `EAGER` ao executar uma consulta.

```
public Pessoa buscaPorTituloComEmailsEager(String nome) {
    String consulta = "select c from Pessoa c
                       join fetch c.emails e
                       where e.titulo = :titulo";

    TypedQuery<Pessoa> query =
              entityManager.createQuery(consulta, Pessoa.class);

    query.setParameter("titulo", titulo);

    Cachorro resultado = null;

    try {
        resultado = query.getSingleResult();
    } catch (NoResultException e) {
        // no result found
    }

    return resultado;
}
```

As vantagens dessa abordagem são:

- Apenas uma consulta é realizada no banco de dados;
- Não existe alteração no modelo;
- Trará apenas os dados necessários;
- Não existe o risco de n+1.

As desvantagens dessa abordagem são:

- Será necessária uma query diferente para cada utilização. Uma query para acessar a lista de `email` e uma outra para trazer uma lista de `pulgas`. Esse é um problema que pode ser minimizado ao se usar uma classe que encapsule as consultas ao banco de dados.

Essa solução pode ser aplicada em ambiente Java EE e Java SE.

Conclusão

Existem diversas soluções para o erro `LazyInitializationException`, mas ambas devem ser utilizadas com cautela. Se adotarmos uma solução e utilizarmos de modo errado, poderíamos estar causando mais mal do que bem para o projeto.

É preciso ter sempre em mente cada vantagem/desvantagem das abordagens vistas aqui e escolher a que melhor se encaixa no projeto.

4.6 Aprenda a tratar o erro: 'cannot simultaneously fetch multiple bags'

É comum termos entidades com mais de um relacionamento. Uma entidade `Usuário` poderia ter uma lista de emails e uma lista de SMSs enviados. Ao utilizar esse tipo de relacionamentos que é que um erro que acontece apenas no Hibernate poderá aparecer.

Esse erro acontece apenas quando uma entity é consultada no banco de dados e ela tem duas ou mais listas configuradas como `EAGER`. Como vimos, o desenvolvedor pode definir as listas como `EAGER` a fim de evitar a `LazyInitializationException` (por exemplo):

```
@Entity
public class Pessoa {

    @OneToMany(mappedBy = "pessoa", fetch = FetchType.EAGER)
    private List<Carro> carros;

    @OneToMany(mappedBy = "pessoa", fetch = FetchType.EAGER)
    private List<Cachorro> cachorros;

    // outras coisas
}
```

Para buscar uma `Pessoa` no banco de dados, o seguinte código seria executado:

```
Pessoa pessoa = entityManager.find(Pessoa.class, 33);
```

Porém, nesse momento, teremos o seguinte erro:

```
javax.persistence.PersistenceException:
org.hibernate.HibernateException: cannot simultaneously fetch
multiple bags
```

O erro acontece quando o Hibernate tenta deixar igual o número de linhas retornadas do banco de dados. Imagine que a entity `Pessoa` com o `id` tenha 1 `Carro` e 2 `Cachorro` associados a ele. Ao realizar a consulta da `Pessoa`, o `Hibernate` tentaria igualar a única linha com resultado de `Carro` com as duas linhas da entity `Cachorro`.

Veja a imagem 4.15 que mostra o problema.

Tabela DOG

id	name	age
1	Red	2
2	Black	3

Tabela CAR

id	name	color
1	Thunder	Green

entityManager.find(Person.class, 33)

person.id	dog.id	dog.name	dog.age	car.id	car.name	car.color
33	1	Red	2	1	Thunder	Green
33	2	Black	3	1	Thunder	Green

Figura 4.15: Resultado da consulta

O problema é que, ao tentar igualar essas linhas, o Hibernate traria resultado repetido.

Existem algumas soluções simples para esse problema comum:

- Utilizar `java.util.Set` ao invés das outras coleções. É a solução mais simples, mas também que pode gerar dor de cabeça para quem usa JSF com a versão igual ou menor que 2.1. Ao alterar de List para Set os métodos hashCode/equals das entidades serão invocados e, com isso, os registros que estariam repetidos não estariam presentes na coleção.

- Como esse é um problema específico do Hibernate, a solução seria usar outra implementação da JPA. Caso a aplicação utilize apenas as anotações/interfaces da JPA, essa alteração seria bem simples e com pouco impacto. Esse é um bom caso que pode justificar a mudança de implementação que está sendo usada.

- Mudar `LAZY`. Desse modo, ao executar o método `find` para buscar uma pessoa, não aconteceria o erro. Porém, é preciso ficar atento para não voltar ao problema da `LazyInitializationException`. Para isso, pode-se passar a fazer a consulta através da JPQL, por exemplo.

Conclusão

Esse erro é comum e pode aparecer bem no começo, quando um desenvolvedor começa a trabalhar com JPA e Hibernate.

As soluções são simples mas é preciso ter bastante cuidado quando for utilizar a opção `EAGER`. O `EAGER` resolve o problema mais facilmente, entretanto, pode fazer com que um objeto com diversas listas venham a ser carregadas em memória.

4.7 Trate o erro: 'could not initialize a collection'

Infelizmente em algumas situações, a JPA pode nos trazer uma exceção cuja mensagem não é clara e por consequência, pouco ajudará para resolver o problema. E é exatamente uma delas que vamos ver nesse capítulo. Dê uma olhada no *stack trace* a seguir:

```
Caused by: com.uaihebert.DBException:
javax.persistence.PersistenceException:
org.hibernate.exception.JDBCConnectionException: could not initialize a
collection: (com.uaihebert.RelatorioVO.sublista
# component(listapk,idDaLista) {idDaLista=523, iofpk=0})
...
...
Caused by: java.sql.SQLRecoverableException: Não serão lidos mais dados
do soquete
```

O erro aqui realmente é exótico e de difícil entendimento. Um dos motivos que podem causá-lo é justamente quando a quantidade de dados a ser trazida do banco de dados é grande e a transação acaba dando erro.

Um erro bem simples para um *stack trace* bem complicado.

Para corrigi-lo, basta paginar a consulta.

4.8 Cuidado para não cair no famoso "efeito n+1"

O efeito n+1 é muito comum ao utilizar o *Open Session In View* ou qualquer outra abordagem que necessite deixar a transação aberta por muito tempo.

Imagine um sistema no qual temos uma rotina que liste todos os empregos de uma pessoa para fazer um processamento. Levando em consideração que a lista de

trabalhos é uma lista do tipo LAZY (todos os detalhes vistos aqui 4.4), após o comando entityManager.find(Pessoa.class, 33); a primeira consulta foi realizada mas ainda não trouxe a lista de trabalhos. Veja o exemplo:

```
String query = "select p from Pessoa p";
// consulta disparada
List<Pessoa> pessoas =
         em.createQuery(query, Pessoa.class).getResultList();
for (Pessoa pessoa : pessoas) {
    // consulta(s) disparada(s)
    List<Trabalho> trabalhos = pessoa.getTrabalhos();

    for (Trabalho trabalho : trabalhos) {
        // consulta(s) disparada(s)
        List<Pessoa> funcionarios = trabalho.getPessoas();
        // faz mais coisas
    }
}
```

O problema começa em dois momentos:

- Caso tenhamos 30 pessoas retornadas na consulta, outras 30 consultas serão realizadas para trazer a lista de trabalhos de cada pessoa;
- Imagine que a entity Trabalho tem uma lista de pessoas que trabalharam lá carregada de modo LAZY também. Só que foi solicitado que esses nomes também fossem listados. Ao fazer trabalho.getPessoas(), outra consulta seria realizada no banco de dados para trazer essa lista. Note que a quantidade de *queries* disparadas cresceria de modo exponencial.

Para evitar esse tipo de efeito, poderiam ser utilizadas soluções vistas no capítulo

.

Conclusão

O efeito N+1 é um problema fácil de cair sem nem perceber. Em geral, o ambiente de desenvolvimento tem uma quantidade de dados bem menor do que em produção, por causa disso, esse efeito N+1 não seria percebido. É preciso ficar atento aos *logs* das consultas sendo executadas.

Escolha a melhor solução que se encaixa em seu sistema e sempre deixe claro para todos os membros da equipe a solução que foi tomada para evitar o N+1.

Capítulo 5

Aprenda os truques da JPQL e domine as consultas da JPA

5.1 Esqueça SQL! Abuse da JPQL

Uma grande facilidade da JPA é justamente a portabilidade. É possível migrar o banco de dados sem necessariamente precisar alterar os códigos escritos. Quem já sofreu dando manutenção a uma aplicação que tem como requisito rodar em diversos bancos de dados diferentes sabe a dificuldade que é.

Imagine um sistema de controle de chamados. Ele registra dados das ligações, dos problemas levantados pelos clientes e o status atual desse chamado. Esse sistema rodará dentro do cliente, e cada cliente pode usar a infraestrutura que quiser, e isso inclui o banco de dados que ele achar melhor. Note que nossa aplicação poderá ter contato com uma grande variedade de 'fabricantes' de banco de dados.

O problema de a mesma aplicação rodar em diversos bancos é justamente a sintaxe de cada query a ser executada. Uma query simples funciona em qualquer banco

de dados: `select * from cachorros`. Mas tudo pode ficar complicado quando precisarmos limitar a quantidade de linhas retornadas em cada consulta. Cada banco de dados faz isso à sua maneira:

```
# MySQL
select * from cachorros LIMIT 10;

# Postgres
select * from cachorros LIMIT 10;

# MS SQLServer
select top 10 * from cachorros;

# Oracle
select * from cachorros where rownum <= 10;
```

Veja que a simples ação de limitar a quantidade de linhas retornadas já geraria bastante dor de cabeça para a aplicação que utiliza diversos bancos. Para contornar esse problema seria necessário criar uma consulta para cada tipo de banco diferente dentro do DAO/Repository, ou pior ainda, fazer uma série de `if` para verificar qual banco está sendo utilizado.

Outra solução muito comum é utilizar um arquivo externo com a sintaxe de cada banco. Nesse caso teríamos que, a cada nova consulta criada na aplicação, replicar as consultas para cada arquivo de consulta de cada banco.

Para ajudar nesse problema de diferentes sintaxes para os diferentes bancos, a JPA vem com a linguagem chamada JPQL (*Java Persistence Query Language*). A sintaxe da JPQL se assemelha muito a uma query normal. O código a seguir fará a mesma coisa que nas 4 consultas já vistas anteriormente:

```
String consulta = "select c from Cachorro c";
TypedQuery<Cachorro> query =
          entityManager.createQuery(consulta, Cachorro.class);
query.setMaxResults(10);
List<Cachorro> resultado = query.getResultList();
```

Vamos analisar por partes essa consulta. `select c from Cachorro c` realiza uma busca no banco de dados trazendo todos os cachorros cadastrados. Note que não existe o `*`, mas existe a letra `c` que informa quais objetos serão retornados, e não campos. Nesse caso, o *alias* `c` foi dado aos objetos do tipo `Cachorro`.

Uma das vantagens de utilizar JPQL é que a JPA já converte o resultado da consulta em objetos. Não é necessário buscar linha por linha e coluna por coluna no objeto como no JDBC. Caso um atributo a mais seja inserido na entidade, a consulta já virá com esse campo populado.

A JPQL na verdade é uma linguagem baseada em objetos. Ao invés de descrevermos como ficará a ligação das tabelas em uma query, escrevemos como os objetos se relacionam. Por isso que, em uma JPQL, é descrito algo como `select p from Pessoa p join p.emails e where e.titulo = 'ABC'`, e a sintaxe da query é feita toda em cima de como o objeto e seus atributos estão escritos.

O método `setMaxResults` limita a quantidade de linhas a serem retornadas. Veja que em nenhum momento foi necessário passar a sintaxe do banco de dados. O comando JPQL acima funcionaria para o MySQL, Oracle e outros mais.

Por fim, o `getResultList()` faz a mágica de converter as linhas retornadas do banco de dados em entidades.

O parâmetro `Cachorro.class` que foi passado no método `createQuery` serve para indicar qual o tipo de retorno da query. Caso o retorno fosse a entidade pessoa, bastaria alterar a query para "select p from Pessoa p" e depois passar como retorno "Pessoa.class".

TypedQuery

A interface `TypedQuery<>` foi criada na versão 2.0 da JPA e permite fazer a consulta sem precisar realizar um cast.

Ao utilizar a JPA 1.0 é necessário utilizar a interface Query, e a consulta acima ficaria como:

```
String consulta = "select c from Cachorro c";
Query query =
    entityManager.createQuery(consulta, Cachorro.class);
query.setMaxResults(5);
List<Cachorro> resultado =
    (List<Cachorro>) query.getResultList();
```

JPQL é uma ferramenta muito poderosa e permite que uma aplicação rode em diversos bancos de dados, sem a necessidade de rescrever a mesma query para cada banco de dados.

HQL nada mais é que a sintaxe do Hibernate para linguagem de banco de dados, com algumas funcionalidades além do que a JPQL, que segue a especificação, possui. Seu funcionamento é o mesmo que o da JPQL, mas funciona apenas com o Hibernate. O Hibernate também implementa suporte ao JPQL.

5.2 Parâmetros com JPQL

A consulta que traz todos os dados pode ser útil em alguns casos, mas, na maioria das vezes, precisamos fazer algum tipo de filtro para buscar informações específicas. Dessa forma, é preciso passar alguns parâmetros para nossa pesquisa.

Para isso, é possível passar valores para a cláusula `where` da sua consulta através da sintaxe da JPQL para parametrização:

```
select c from Cachorro c where c.nome = :nome

select c from Cachorro c where c.idade = :idade

select c from Cachorro c where c.idoso = false
```

Chamamos essa abordagem de parametrização de *parâmetros nomeados*. Para passar o valor para a consulta, basta fazer como:

```
String consulta = "select c from Cachorro c where c.idade = :idade";
TypedQuery<Cachorro> query =
        entityManager.createQuery(consulta, Cachorro.class);
query.setParameter("idade", 33);
```

Ao utilizar o método `setParameter` estamos informando à JPA que o parâmetro deve ser substituído.

Evite concatenar variáveis nas consultas

Repare na consulta a seguir:

```
select c from Cachorro c where c.nome = " + nome
```

Ao concatenar um atributo qualquer a uma query, um usuário malicioso poderia inserir um script SQL aí. Esse tipo de ataque é conhecido como *SQL Injection*.

Algumas pessoas pensam que pelo fato de estarem usando JPQL nada de mal pode acontecer, **e estão muito erradas**. Um hacker que tenha o conhecimento de JPQL poderia facilmente inserir um código em JPQL ali ou em qualquer outra parte do sistema.

Para ter ideia de como uma pessoa poderia utilizar a query em seu favor, bastaria que o hacker, ao invés de escrever puramente o nome, escrevesse algo como: jose '' OR '' joao. Desse modo ele dispararia uma pesquisa por dois nomes e não um apenas, pois ele adicionou a condição 'OR' do SQL no campo nome.

Tenha bastante cuidado, nunca concatene sua query. Seja com SQL normal ou com JPQL/HQL.

Quando trabalhamos com datas em banco de dados podemos encontrar os valores em formato de data apenas ('10-10-2010'), hora apenas ('22:10:10 000'), data + hora ('10-10-2010 22:10:10 000') e em alguns casos milissegundos (1000 que daria 1s). Veja que teríamos que fazer um *parse* para cada tipo de data com que fôssemos trabalhar. Outra facilidade da JPA são consultas com datas:

```
String consulta = "select c from Cachorro c
                   where c.dataRegistro < :data";
TypedQuery<Cachorro> query =
        entityManager.createQuery(consulta, Cachorro.class);
Data dataAtual = new Date();
query.setParameter("data", dataAtual);
```

Muitas vezes temos o objeto pronto, mas queremos apenas passar seu id para a comparação. Nesse caso, podemos passar o objeto como parâmetro e automaticamente a JPA comparará o id da entity passada no parâmetro.

```
String consulta = "select c from Cachorro c where c.dono = :dono";
TypedQuery<Cachorro> query =
        entityManager.createQuery(consulta, Cachorro.class);
query.setParameter("dono", dono);
```

5.3 Navegações nas pesquisas

Por ser uma linguagem de pesquisas baseada na orientação a objetos, trabalhar com relacionamentos é feito através da navegação pelos objetos. O mesmo vale para ordenar os resultados das pesquisas por JPQL igual a uma consulta SQL. É preciso levar em consideração alguns detalhes que veremos abaixo.

5.3.1 Join

É possível realizar navegação de entidades para realizar `join` em coleções ou atributos. Veja os exemplos:

```
#1
select p from Pessoa p join p.carros
#2
select p from Pessoa p left join p.carros
#3
select p from Pessoa p left join fetch p.carros
```

O primeiro `select` mostra um simples `join`. Note que o join é realizado com nome do atributo dentro da classe e não o nome da tabela. No segundo, é utilizada a opção `left join` que tem a mesma funcionalidade em *queries* SQL.

A consulta #3 é executada com a opção `fetch`, e faz com que o atributo *LAZY* venha como *EAGER*. Essa é uma das soluções vistas no capítulo 4.8 para evitar a `LazyInitializationException`.

E para utilizar as consulta com `join`, não precisamos fazer nada de diferente em nosso código:

```
String consulta = "select p from Pessoa p join p.carros";
TypedQuery<Pessoa> query =
          entityManager.createQuery(consulta, Pessoa.class);
List<Pessoa> resultado = query.getResultList();
```

5.3.2 Faça ordenações

É possível também ordenar a consulta a partir de um campo desejado.

```
select p from Pessoa p order by p.nome
```

A consulta traria todas as pessoas do banco de dados ordenadas pelo nome. Novamente, a sintaxe da JPQL tenta ser o mais próxima possível dos SQLs.

5.3.3 Navegando pelos relacionamentos

Outra funcionalidade das consultas é a possibilidade de navegar dentro dos relacionamentos. É nessa hora que podemos ver o conceito de *JOIN Explícito* e *JOIN Implícito*.

O *JOIN Explícito* é o exemplo já visto neste capítulo no qual o comando a seguir é utilizado nas consultas: `join p.cachorros`. Já o *JOIN Implícito* acontece quando temos uma navegação em um relacionamento.

Para entender melhor a diferença entre um *JOIN Explícito* e um *JOIN Implícito* é que o *JOIN Explícito* é quando temos a palavra *JOIN* escrita na consulta; um *JOIN Implícito* é quando temos a navegação através de atributos como por exemplo: `pessoa.carro.nome`.

É preciso apenas ter cuidado ao escolher qual o tipo de *JOIN* utilizar. Tome por exemplo a entity `Pessoa`. Imagine que precisamos encontrar pessoas com o carro da cor vermelha.

```
@Entity
public class Pessoa{

    ...

    @OneToOne
    private Carro carro;

    ...
}
```

Para realizar um *JOIN Implícito* entre `Pessoa` e `Cachorro` basta fazer como a seguir:

```
select p from Pessoa p where p.carro.cor = 'Vermelha'
```

A query resultará em um *JOIN* a ser realizado na tabela da entidade `Carro`. No entanto, não é possível fazer a mesma funcionalidade de consulta que fizemos em uma lista.

```
select p from Pessoa p where p.carros.cor = 'Vermelha'
```

Para navegar em uma lista é necessário declarar o *JOIN* a ser utilizado, como o que segue:

```
select p from Pessoa p join p.namoradas ex
     where ex.nome = 'Josefina Antonieta'
```

5.4 Funções Matemáticas

Veremos nesse capítulo funções matemáticas que facilitam na hora de gerar relatórios ou em consultas para buscar dados no banco de dados. Funções matemática que são muito úteis na hora de gerar um relatório ou realizar algum processamento. Para calcular o salário do funcionário é necessário fazer um somatório de todos os dias trabalhados, assim como somar todas as horas extras e outros cálculos.

[subsection Faça a soma com a função SUM()]

Imagine que foi solicitada a somatória das notas de um aluno em específico, caso, o aluno que tem o ID = 33. A função `SUM()` serve para somar valores dos atributos de uma entity. Veja a entity abaixo:

```
package com.model;

@Entity
public class Aluno {
    private int idade;
    private String nome;
    private long notaTotal;

    @ElementCollection
    private List<Integer> notas;

    public Aluno() {

    }

    // outras coisas
}
```

Primeiramente vamos trazer a soma das notas do aluno de id 33:

```
String consulta = "SELECT sum(n) FROM Aluno a " +
                  "join p.notas n where a.id = :id";
TypedQuery<Number> query = em.createQuery(consulta, Number.class);
query.setParameter("id", 33);
System.out.println(query.getSingleResult());
```

Note que o resultado será o somatório das notas de um `Aluno` em específico.

Imagine agora um relatório no qual deve ser exibido o nome, idade e o total da nota de todos os alunos cadastrados. Em qualquer consulta JDBC precisamos buscar o objeto e populá-lo manualmente, algo que dá muito trabalho.

```
ResultSet result = // pega o resultado da consulta
String nome = resultSet.getString(0);
int idade = resultSet.getInt(1);
long notaTotal = resultSet.getLong(2);

Aluno aluno = new Aluno(nome, idade, notaToTal);
```

Note que, quando usamos JDBC, teremos mais linhas de código Java para receber esse valor, e colocar o `resultSet.getLong(2)` dentro do construtor ficaria ilegível caso o número argumentos fosse grande. Para tornar essa ação mais simples com JPA, vamos criar um construtor e um campo na entity `Aluno`:

```
package com.model;

@Entity
public class Aluno {
    @Transient
    private long notaTotal;

    public Aluno(String nome, int idade, long notaTotal) {
        this.nome = nome;
        this.idade = idade;
        this.notaTotal = notaTotal;
    }

    // outras coisas
}
```

Veja que adicionamos um campo chamado `long notaTotal` e um construtor na classe. Note que a classe já tinha o construtor padrão declarado `public Aluno(){}`. O construtor padrão é obrigatório para que uma entity seja corretamente utilizada pela JPA.

> **@Transient**
>
> A anotação @Transient informa que esse campo não deve ser persistido pelo JPA, desse modo podemos ter sempre o valor atualizado dinamicamente.

Vamos agora executar a query que realizará o `SUM()` trazendo todas os alunos e a soma de suas respectivas notas:

```
String consulta = "SELECT NEW com.model.Aluno(a.nome,a.idade,sum(n))"+
                  "FROM Aluno a join a.notas n " +
                  "GROUP BY a.nome, a.idade";
TypedQuery<Aluno> query = em.createQuery(consulta, Aluno.class);
List<Aluno> result = query.getResultList();
```

Com a consulta acima, teríamos como resultado objetos do tipo `Aluno` com o total de notas de cada um corretamente preenchido. [/subsection]

5.4.1 Calculando mínimos e máximos

Da mesma maneira que SQL possui as funções para extrair o valor mínimo e máximo de um resultado, geralmente chamamos de `min` e `max`, com a JPQL também é possível:

```
String consulta = "SELECT max(a.idade), min(a.idade) FROM Aluno a";
TypedQuery<Object[]> query = em.createQuery(consulta, Object[].class);
Object[] result = query.getSingleResult();
```

Quando temos uma consulta em que não é informado qual o objeto a ser retornado, teremos como resultado um array de objeto (Object[]). O resultado é um `Object[]` no qual a ordem das funções será respeitada. Como o `max(a.idade)` foi descrito primeiro, seu resultado estará na primeira posição do *Array*, `result[0]`, enquanto o `min(a.idade)` está na segunda posição do *Array*: `result[1]`.

5.4.2 Contando resultados

Para realizar uma contagem de quantos registros existem no banco de dados para uma consulta, basta fazer como a seguir:

```
String consulta = "SELECT COUNT(a) FROM Aluno a";
TypedQuery<Number> query = em.createQuery(consulta, Number.class);
Number result = query.getSingleResult();
```

5.4.3 Outras funções: MOD, SQRT e AVG

Com a função `MOD` é possível receber o resto de uma divisão; já com a função `SQRT` é possível determinar a raiz quadrada de um dado valor; e com o `AVG` é possível calcular a média de um determinado campo.

5.5 Funções String

Quando estamos a trabalhar com dados não temos como prever todas as situações; um usuário poderia cadastrar um nome como JOSÉ e outro como jose. É possível encontrar também alguém querendo pesquisar apenas por pessoas que contenham a palavra *oliveira* no nome, mas independente de se é caixa alta ou baixa. É difícil prever todas as situações possíveis e pior seria se tivéssemos que escrever códigos Java para tratar todos os casos.

A JPA também provê funções para facilitar o trabalho com `String`. A seguir é possível ver alguns exemplos de como utilizar diversas dessas funções:

```
#1
select p from Pessoa p where p.nome = trim(:nome)

#2
select lower(p.nome) from Pessoa p where upper(p.nome) = 'JOSE'
```

A função `trim(:nome)` eliminará os espaços vazios a esquerda e a direita do campo, mas não elimina o espaço em branco que se encontra em uma `String`. O valor `"Jose de Arimatéia "` ao passar pelo método `trim()` ficaria `"Jose de Arimatéia"`.

A função `lower(p.nome)` deixará todos os *cases* minúsculos; de `JOSE de ARIMatéia` irá para `jose de arimatéia`. Já a função `upper(p.nome)` fará

o contrário da função `lower(p.nome)`; toda a String terá o *cases* maiúsculos, de `JOSE de ARIMatéia` irá para `JOSE DE ARIMATÉIA`.

Existe também a função `LENGTH` que retorna o tamanho de uma `String`:

```
String consulta = "SELECT LENGTH(p.name) "+
                  "FROM Aluno p where p.idade = 33";
TypedQuery<Number> query = em.createQuery(consulta, Number.class);
Number result = query.getSingleResult();
```

Existem casos em que apenas um pedaço de um texto é necessário para nossa consulta. Se fosse para gerar um adesivo com apenas as 3 primeiras letras de um aluno, extrair as 4 últimas letras de um código de algum produto como 123OQIWL.

Para finalizar, vamos falar da função chamada `SUBSTRING`, que permite extrair o pedaço de uma String do banco de dados. Veja a consulta:

```
String consulta = "SELECT SUBSTRING(p.nome, 1, 3) " +
                  " FROM Aluno p where p.nome = 'John'";
TypedQuery<String> query = em.createQuery(consulta, String.class);
String result = query.getSingleResult();
System.out.println(result); // imprimirá Joh
```

A função `SUBSTRING` pode ser utilizada também após o `WHERE` como pode ser visto a seguir:

```
SELECT a.nome FROM Aluno a where SUBSTRING(a.nome, 1, 3) = 'Joh'
```

5.6 Agrupadores - group by e having

Ao utilizar funções como `SUM` e `COUNT` pode ser necessário agrupar o resultado, ação normal até com SQL nativo.

Essas funções são muito boas para quando você quer tirar média ou realizar somatórios. Por exemplo, caso fosse necessário saber o total de alunos reprovados em determinada matéria, ou então, o somatório de todos os impostos pagos em determinada empresa.

Ao utilizar a função `SUM()` para trazer a soma juntamente com outros campos, seria necessário agrupar o resultado com o `group by`.

Vamos começar somando todas as notas obtidas dos alunos:

```
String consulta = "SELECT SUM(n.total)" +
              "FROM Aluno p join p.notas n"
TypedQuery<Long> query = em.createQuery(consulta, Long.class);
```

Foi possível ver como extrair o somatório das notas ao utilizar a função SUM da JPA. Não é comum trazer apenas o somatório em uma consulta, mas sim o somatório e outros campos. Veja o exemplo:

```
String consulta = "SELECT NEW com.model.Aluno(" +
                          "p.nome, p.idade, SUM(n)) " +
             "FROM Aluno p join p.notas n " +
             "GROUP BY p.name, p.age";
TypedQuery<Aluno> query = em.createQuery(consulta, Aluno.class);
```

A consulta mostrada traz a soma das notas, o nome e a idade do aluno para dentro de um objeto `Aluno`. O `group by` é necessário, pois estamos realizando uma operação matemática em que a função é executada juntamente com outros valores.

Imagine que precisamos apenas dos alunos que tenham a nota acima de 80. É nesse momento que podemos utilizar o `HAVING`. Veja abaixo como ficaria nossa consulta:

```
SELECT NEW com.model.Aluno(p.name, p.age, SUM(n))
FROM Aluno p join p.notas n
GROUP BY p.name, p.age
HAVING SUM(n) > 80
```

Com a função `HAVING`, um filtro seria feito e traria apenas os alunos com as notas acima de 80.

5.7 Condições para comparações

Existem diversas condições que podem nos ajudar na hora de definir as comparações da pesquisa. Podemos definir condições como as cláusulas para filtrar uma consulta após o `WHERE` que pode ou não chamar uma função, por exemplo `where nome = 'Minhoca'`. Como fazer para filtrar entidades que não têm determinada lista vazia, ou valores que serão encontrados em uma subconsulta?

Veremos diversas funções e suas utilizações a seguir.

5.7.1 Restrinja pesquisas por uma lista com o IN

É comum termos que procurar por algum registro em uma tabela quando temos uma faixa de valores como entrada de dados. Procurar todas as pessoas com idade 10, 15 e 20 ou até mesmo todos os produtos que possuem a categoria A, B e C.

É possível informar uma lista de valores para a JPA e utilizar a função `IN` na hora da consulta:

```
List<Integer> numeros = new ArrayList<Integer>();
numeros.add(1);
numeros.add(2);
numeros.add(3);

String consulta =
         "SELECT a.nome FROM Aluno a where a.idade in (:id)";
TypedQuery<String> query = em.createQuery(consulta, String.class);
query.setParameter("id", numeros);
List<String> result = query.getResultList();
```

A função `IN` pode receber uma `List` como parâmetro, mas somente na versão 2.0 da JPA.

Outro caso em que poderíamos utilizar o `IN` é quando a lista de valores já está no banco de dados:

```
SELECT a1 FROM Aluno a1 where a1 IN (
         select a2 from Cachorro c
         join d.aluno a2 where c.peso > 60
)
```

A subquery busca por todos os cachorros que têm o peso maior que 60. Uma vez que a subquery encontre o resultado, eles irão para a variável `a2`. A query principal fará a procura por todos os alunos que se encontram em `a1` comparando com o resultado de `a2`.

5.7.2 Evite repetições com DISTINCT

Existem casos em que precisamos de resultados não repetidos. Muitas vezes os valores retornados em uma consulta na qual se usa o *JOIN* têm repetição.

Um modo de evitar esse problema é utilizando o comando `distinct` nas consultas. Veja a consulta abaixo:

```
SELECT a FROM Aluno a
join a.cachorros c where c.peso > 0
```

A consulta anterior faria com que houvesse resultados repetidos dentro de um `List<Aluno>`. Para resolver isso, podemos adicionar a palavra `DISTINCT` na consulta:

```
SELECT DISTINCT a FROM Aluno a
join a.cachorros c where c.peso > 0
```

Assim estamos dizendo que os alunos `a` devem ser distintos, ou seja, não repetidos.

5.7.3 Listas e valores vazios com EMPTY e NULL

Existem funções que nos ajudam a tratar coleções dentro de entidades. Imagine situações com funcionários que não têm marcações de ponto no dia ou pessoas cadastradas sem nenhum e-mail.

É possível verificar se uma lista está ou não vazia com a condição `EMPTY`:

```
# 1
select p from Pessoa p where p.cachorros is empty

# 2
select p from Pessoa p where p.cachorros is not empty
```

A condição `EMPTY` deve ser utilizada com coleções. Caso fosse necessário saber se um atributo que não seja coleção esteja vazio, basta fazer a comparação com `NULL` ao invés de `EMPTY`:

```
select p from Pessoa p where p.namorada is null
```

5.7.4 Pesquise por intervalos com BETWEEN

Em um sistema é comum procurar um resultado em um intervalo de valor, seja ele de datas ou numérico. Em um sistema de pet shop, por exemplo, poderíamos buscar por todos os cachorros que não receberam vacina no último semestre para receberem um convite.

Para consultas utilizando espaço de tempo, utilizamos a condição `BETWEEN`:

```
select c from Cachorro c
where c.dataNascimento between :dataInicial and :dataFinal
```

É possível também realizar essa pesquisa com números, por exemplo, uma busca por cachorros com idade entre 1 e 3 anos:

```
select c from Cachorro c
where c.idade between 1 and 3
```

E para finalizar, é possível utilizar `between` para buscar valores textuais, como nome, apelido etc. Para buscar todos os cachorros que tenham o nome começando entre 'b' e 'd':

```
select c from Cachorro c
where c.nome between 'b' and 'd'
```

No exemplo, o nome 'calango' seria retornado na pesquisa.

5.7.5 Busca por trechos de texto com LIKE

Para facilitar pesquisas e melhorar a utilização do sistema, é comum permitir que o usuário faça pesquisas por pedaços de uma String. Em vez de ter que digitar o nome completo de uma `Pessoa` em um hospital, por exemplo '*Andre Luiz da Costa Lemos*', ao utilizar a condição `LIKE` e digitando apenas `Andre Luiz` o registro será localizado no banco de dados:

```
String consulta = "select p from Pessoa p where p.nome like :nome";
TypedQuery<Pessoa> query = em.createQuery(consulta, Pessoa.class);
query.setParameter("nome", "%" + nome + "%");
List<Cachorro> resultado query.getResultList();
```

Um detalhe interessante é que o símbolo `%` é passado juntamente com o parâmetro e não fica estático dentro da JPQL.

5.7.6 Verifique se um elemento existe com o MEMBER OF

Temos a necessidade de procurar por `Alunos` que tenham determinado `Cachorro`, sendo que esse `Cachorro` é um objeto selecionado na tela. Em vez de ter que escrever um `JOIN` para comparar `ID`, bastaria utilizar o comparador `MEMBER OF`:

```
select a from Aluno a where :cachorro member of a.cachorros
```

Isso seria equivalente a saber se um atributo existe em uma coleção, algo como `aluno.getCachorros().indexOf(cachorro);`.

5.7.7 Operações em listas com EXISTS, ANY, SOME e ALL

É possível trazer resultado utilizando subqueries para validar a existência de determinados valores. Imagine que precisamos buscar todos os alunos que moram na mesma casa, mas temos apenas o número da casa. Essa tarefa é fácil com as condições `EXISTS`, `ANY` e `SOME`:

```
select a from Aluno a
where exists (select e from a.endereco e
              where e.numeroCasa = :numeroCasa)

select a from Aluno a
where a.endereco = any (select e from Endereco e
                        where e.numeroCasa = :numeroCasa)

select a from Aluno a
where a.endereco = some (select e from Endereco e
                        where e.numeroCasa = :numeroCasa)
```

Na prática, todas as funções têm a mesma funcionalidade, tendo apenas o nome diferente.

Para comparar o valor de um atributo de uma lista podemos utilizar a condição ALL:

Imagine um caso no qual precisamos buscar cachorros de um determinado peso, mas realizando uma segunda consulta que poderia fazer outras verificações, como a idade do cachorro:

```
select p from Pessoa p
where p.cachorros is not empty
and :peso < all
    (select c.peso from p.cachorros c where c.idade = 10)
```

O peso só retornará TRUE caso todos os cachorros da pessoa passem na verificação. Um detalhe é que, caso a lista esteja vazia, o valor retornado será sempre TRUE. Para evitar esse problema é necessário validar se a lista não está vazia.

5.7.8 Use CONCAT para concatenar Strings

É possível fazer a concatenação de duas String através da função CONCAT:

```
select p from Pessoa p
where p.nome = concat(:primeiraPalavra, :segundaPalavra)
```

O CONCAT também pode ser utilizado para tratar o resultado de uma consulta como adiante:

```
select concat(p.nome, p.sobreNome) from Pessoa p
```

Dessa maneira, será devolvido o nome junto do sobrenome para cada pessoa.

5.7.9 Verifique a posição de um texto com o LOCATE

Existem casos em que ao trabalhar com String precisamos saber apenas se determinado texto contém algum valor. Por exemplo, se algum registro de log no banco de dados tem a palavra `ERROR`; outro exemplo seria um sistema que busca por familiares quer localizar pessoas cadastradas que tenham o sobrenome oliveira.

A função `LOCATE` verifica a posição de um determinado texto dentro de outro:

```
select p from Pessoa p
where locate(p.nome, :value) > 0
```

5.7.10 Identifique o tamanho de listas com o SIZE

Existe também uma função que permite verificar o tamanho de uma lista. A função `SIZE` pode ser utilizada tanto como uma cláusula `WHERE` como resultado de uma pesquisa:

```
select p from Pessoa p
where size(p.cachorros) = :quantidade

select size(p.cachorros) from Pessoa p
where p.nome = :nomePessoa
```

5.8 Trabalhando com data e hora atual

Em alguns casos necessitamos manipular datas no servidor em nossas consultas. Para comparar a data do servidor com atributos de uma entity basta fazer como a seguir:

```
select c from Cachorro c
where c.dataNascimento < CURRENT_DATE
```

A pesquisa acima fará a comparação entre a `dataNascimento` com a data atual do servidor, por exemplo: `13/12/2011`. É possível pesquisar apenas pela hora utilizando `CURRENT_TIME` ou por data e hora com `CURRENT_TIMESTAMP`.

5.9 Buscando apenas um resultado na consulta

Um modo simples de buscar uma única entidade em uma consulta é utilizar o método `getSingleResult`, como o que segue:

```
String consulta = "select c from Cachorro c where c.nome = 'Minhoca'"
TypedQuery<Cachorro> query =
      entityManager.createQuery(consulta, Cachorro.class);
Cachorro cachorro = query.getSingleResult();
```

O código funcionaria sem problema algum, correto? Sim e não! O método `getSingleResult` pode lançar duas exceções que podem pegar o desenvolvedor desprevenido.

A primeira exceção que pode acontecer é caso tenha mais de um registro retornado pela consulta. Imagine que a consulta retornasse dois cachorros com o nome `Minhoca`. Então seria lançado um erro parecido com o seguinte:

```
Exception in thread "main" javax.persistence.NonUniqueResultException:
                        result returns more than one elements
at org.hibernate.ejb.QueryImpl.getSingleResult(QueryImpl.java:287)
```

A segunda exceção que pode acontecer é caso nenhum resultado seja encontrado no banco de dados. Imagine que nenhum cachorro cadastrado com o nome de `Minhoca` seja encontrado. Uma exceção como a que segue será lançada:

```
Exception in thread "main" javax.persistence.NoResultException:
                                No entity found for query
at org.hibernate.ejb.QueryImpl.getSingleResult(QueryImpl.java:280)
```

Então fique atento: ao invés de retornar `null` caso nenhum resultado seja encontrado, a JPA lançará uma exceção.

Existem desenvolvedores que, como solução, buscam a lista de resultados com o método `getResultList` e a manipulam através do `isEmpty` para saber se há algum conteúdo ou não.

5.10 Criando objetos com o retorno de consultas

É muito comum termos relatórios em nossos sistemas que vez ou outra pedem dados que estão além de uma entity. Imagine um relatório da entidade `Pessoa`, no qual é necessário exibir o nome da pessoa e o total de cachorros. Poderíamos criar uma classe apenas de leitura para trazer essas informações, que ficaria como:

```
public class PessoaComTotalCachorroVO {
    private Pessoa pessoa;
    private int totalCachorro;
```

```
    public PessoaComTotalCachorroVO(Pessoa pessoa, int totalCachorro) {
        this.pessoa = pessoa;
        this.totalCachorro = totalCachorro;
    }

    public Pessoa getPessoa() {
        return pessoa;
    }

    public void setPessoa(Pessoa pessoa) {
        this.pessoa = pessoa;
    }

    public int getTotalCachorro() {
        return totalCachorro;
    }

    public void setTotalCachorro(int totalCachorro) {
        this.totalCachorro = totalCachorro;
    }
}
```

Note que a classe `PessoaComTotalCachorroVO` não é uma entidade, apenas uma classe normal, que carregará alguns valores do banco de dados para a aplicação. Por isso, o sufixo `VO`, de *Value Object*. Essa classe tem uma referência para a entity `Pessoa`, mas não é uma entidade. Seu estado não é gerenciado pelo `EntityManager`. O pattern *Data Transfer Object* (DTO) também pode aparecer em alguns desses casos. Diversos desenvolvedores preferem evitar esses sufixos, para deixar o código mais simples de ler.

E para criar o relatório bastaria executar uma consulta como:

```
String consulta =
    "select new com.uaihebert.report.PessoaComTotalCachorroVO(p, " +
                                "size(p.cachorros)) from Pessoa p group by p";
TypedQuery<PessoaComTotalCachorroVO> query =
        em.createQuery(consulta, PessoaComTotalCachorroVO.class);
List<PessoaComTotalCachorroVO> resultList = query.getResultList();

for (PessoaComTotalCachorroVO reportVO : resultList) {
    System.out.println(reportVO.getPessoa().getNome() +
```

```
                        " " +
                        reportVO.getTotalCachorro());
}
```

Note que a consulta utilizada realiza o comando `new` para instanciar. É necessário que a classe tenha um construtor com os exatos valores retornados na consulta.

Essa prática é muito boa para facilitar a criação de relatórios que já venham com os valores todos prontos de uma consulta.

Capítulo 6

Alternativas às consultas: Named Queries e Queries nativas

6.1 Organizando consultas com NamedQuery

Por cada empresa que passamos, é comum encontrarmos projetos enormes, com diversas consultas para cada requisito. Imagine um sistema em que um desenvolvedor precise consultar a data de nascimento de uma pessoa para exibir sua idade. Em outro momento, outro desenvolvedor precisará buscar pessoas pela data de nascimento para calcular o valor do plano de saúde. Note que se não tivermos um lugar para concentrar essas consultas facilmente teremos consultas repetidas em nossa aplicação.

Para ajudar a evitar repetição de consultas, podemos utilizar a chamada `NamedQuery`; uma `NamedQuery` deve ser declarada em uma entity e seu nome deve ser único. Veja a seguir como declarar uma `NamedQuery`:

```
@Entity
@NamedQuery(name = Aluno.BUSCAR_POR_NOME,
```

```
            query = "select a from Aluno a where a.nome = :nome")
public class Aluno {

    public static final String BUSCAR_POR_NOME = "Aluno.BuscarPorNome";
    // outras coisas
}
```

No código da classe `Aluno`, é possível ver que `@NamedQuery` tem dois atributos: `name` e `query`. `name` define um nome único dado à `NamedQuery`. Se uma `NamedQuery` for nomeada com o nome de outra `NamedQuery` já existente, uma exceção será lançada. `query` define a `JPQL` a ser utilizada.

As vantagens da `NamedQuery` começam em sua utilização:

```
TypedQuery<Aluno> query =
    entityManager.createNamedQuery(Aluno.BUSCAR_POR_NOME, Aluno.class);
query.setParameter("nome", nome);
Aluno aluno = query.getSingleResult();
```

Veja que não foi necessário digitar a JPQL novamente, bastou fazer referência ao seu nome através do método `createNamedQuery`. Para o problema citado no começo do capítulo, bastaria o desenvolvedor verificar se a JPQL já existia na entity e reutilizá-la onde quisesse.

Para declarar mais de uma `@NamedQuery`, basta utilizar a anotação `@NamedQueries`:

```
@Entity
@NamedQueries({
    @NamedQuery(name = Aluno.BUSCAR_POR_NOME,
                query = "select a from Aluno a where a.nome = :nome"),
    @NamedQuery(name = Aluno.BUSCAR_POR_IDADE,
                query = "select a from Aluno a where a.idade = :idade")
})
public class Aluno {

  public static final String BUSCAR_POR_NOME = "Aluno.BuscarPorNome";
  public static final String BUSCAR_POR_IDADE = "Aluno.BuscarPorIdade";
    // outras coisas
}
```

A `@NamedQueries` é utilizada para agrupar todas as `@NamedQuery` da entity.

Outra vantagem da `NamedQuery` é que a sintaxe da consulta é validada quando o método `Persistence.createEntityManagerFactory()` é executado, que geralmente é no início da aplicação.

Para ter uma melhor organização das consultas, uma boa prática é utilizar constantes para dar o nome da consulta.

```
@Entity
@NamedQuery(name = Aluno.BUSCAR_POR_NOME,
            query = "select a from Aluno a where a.nome = :nome")
public class Aluno {
    public static final String BUSCAR_POR_NOME = "Aluno.BuscarPorNome";

    // ...
}
```

Com isso, evitamos escrever o nome errado, seja na hora de definir a query ou na hora de executá-la. O erro seria descoberto apenas em tempo de execução, e para quem tem um deploy lento isso poderia ocasionar uma boa perda de tempo.

Uma outra boa prática seria o modo de nomear a `NamedQuery`. Se o nome da `NamedQuery` fosse apenas `BuscarPorNome`, em alguma outra entity poderia ter uma `NamedQuery` com o mesmo nome. Para evitar esse tipo de situação é considerada boa prática nomear a `NamedQuery` com prefixo: `nome da entidade.`.

Por outro lado, caso tenha muitas queries, a sua entidade pode ficar muito poluída e, por consequência, ficar ruim de se dar manutenção.

6.2 Quando há algo muito específico, utilize Query nativa

Ao utilizar JPA é necessário ter em mente que, um dos motivos pelo qual a JPA foi desenvolvida é: permitir a portabilidade entre bancos. A *'desvantagem'* é que recursos e otimizações específicas do banco de dados não podem ser utilizados.

Caso dê para garantir que sua aplicação será utilizada em apenas um banco de dados, então é possível aproveitar esses recursos específicos através da `NativeQuery`, que permite que funções nativas do banco de dados sejam utilizadas.

Imagine um sistema em que a senha do aluno será salva em MD5. Ao tentar fazer o login, nós devemos comparar a senha que o usuário digitou com a senha salva. Para facilitar, vamos utilizar a função `md5` do banco postgres:

```
String consulta = "SELECT * FROM Aluno a where
                  md5(a.senha) = :senha";

Query query = entityManager.createNativeQuery(consulta, Aluno.class);
query.setParameter("senha", senha);
Aluno aluno = (Aluno) query.getSingleResult();
```

Veja no exemplo que a função `md5(...)` foi chamada de dentro da consulta. A sintaxe de uma `NativeQuery` é a mesma do banco em que está sendo executada, lembrando que para criá-la, o método chamado é `createNativeQuery`. Perceba que ele não retorna uma `TypedQuery`, portanto será necessário fazer `cast` do objeto retornado pela consulta.

Apesar da facilidade de uso, é preciso ficar atento a uma eventual mudança de banco de dados, pois podem ocorrer erros nas consultas, já que elas foram pensadas em um tipo de banco de dados específico.

6.3 Devolva resultados complexos com queries nativas

Quando utilizamos consultas nativas é normal ter muitas informações sendo retornadas, até mesmo mais informações do que apenas uma entity.

```
select *
from pessoa p
left join cachorro c on p.id = c.pessoa_id
```

A query acima traria os dados de `Pessoa` e `Cachorro` ao mesmo tempo. Porém, a JPA não sabe como retornar o resultado dessa consulta. Não existe nenhum objeto que mapeie ao mesmo tempo para cachorro e pessoa. Para resolver esse problema temos a anotação `@SqlResultSetMapping`. Veja como utilizar:

```
@Entity
@SqlResultSetMapping(name= Pessoa.PESSOA_CACHORRO_MAPPING,
    entities={
       @EntityResult(entityClass=Pessoa.class),
       @EntityResult(entityClass=Cachorro.class,
       fields={
           @FieldResult(name="id", column="CACHORRO_ID")
       }
```

```
)
public class Pessoa {
    // outras coisas
    public static final String
                PESSOA_CACHORRO_MAPPING = "Pessoa.comCachorroMapping";
}
```

Vamos analisar o código:

- `@SqlResultSetMapping` indica à JPA que um mapeamento especial deve ser utilizado quando trabalhamos com consultas nativas;
- `name` indica um nome único que deve ser dado ao mapeamento especial;
- `@EntityResult` define as entidades que serão retornadas na consulta;
- `@FieldResult` serve para diferenciar campos com mesmo nome. Imagine que na tabela de pessoa existe a coluna ID e na tabela cachorro também existe a coluna ID, como a JPA saberia qual ID pertence a qual entity? Veremos mais à frente como fazer a diferença na hora da consulta.

E para utilizar o mapeamento acima, basta fazer como a seguir:

```
String consulta = "select p.*, c.id as CACHORRO_ID, c.nome" +
              " from pessoa p join cachorro c on c.pessoa_id = p.id" +
              " where p.id = 33";

Query query = em.createNativeQuery(consulta, PESSOA_CACHORRO_MAPPING);

Object[] resultado = (Object[]) query.getSingleResult();

Pessoa pessoa = (Pessoa) resultado[0];
Cachorro cachorro = (Cachorro) resultado[1];
```

Veja na consulta que o `id` do `Cachorro` recebeu o apelido de `CACHORRO_ID`; esse apelido é criado para facilitar o mapeamento `@FieldResult` utilizado na entity; caso existissem dois campos com mesmo nome, seria necessário dar nomes diferentes (como feito com o `CACHORRO_ID`).

É possível também realizar mapeamento de valores que não pertencem a uma entity. Imagine que existe a necessidade de buscar cada pessoa e sua respectiva quantidade de cachorros. A query seria mapeada como o seguinte:

```
@Entity
@SqlResultSetMapping(name= PESSOA_TOTAL_CACHORRO,
     entities={@EntityResult(entityClass=Pessoa.class)},
     columns={@ColumnResult(name="totalCachorro")}
)
public class Pessoa {
    // outras coisas
    public static final String
                PESSOA_TOTAL_CACHORRO = "Pessoa.comTotalCachorro";
}
```

A consulta trará apenas a entity `Pessoa`, juntamente com uma coluna contendo o total de cachorros de cada `Pessoa`. Veja como ficará:

```
String consulta = "select p.id, p.nome, count(0) as totalCachorro " +
"from pessoa p join cachorro c on p.id = c.person_id " +
"where nome = 'José de Arimatéia' " +
"group by p.id, p.nome";

Query query = entityManager.createNativeQuery(consulta,
                    Pessoa.PESSOA_TOTAL_CACHORRO);

Object[] result = (Object[]) query.getSingleResult();

Pessoa pessoa = (Pessoa) result[0];
BigInteger totalCachorro = (BigInteger) result[1];
```

Veja a facilidade que a JPA nos traz com os mapeamentos complexos. É possível trazer resultados que não são *Entities* sem problema algum. A vantagem desse mapeamento sobre o JDBC é que, apesar de o resultado vir dentro de um `Object[]`, a entidade já vem instanciada com os valores. Caso fosse JDBC, teríamos que popular campo a campo o resultado.

Capítulo 7

Entenda as queries programáticas com Criteria

Quando temos uma tela com diversas opções onde é possível fazer consultas com parâmetros variados realmente fica complicado de montar a consulta.

Imagine uma tela que tenha onde diversas informações poderiam ou não participar de uma consulta. Note que o botão de pesquisa pode ser acionado com uma ou mais opções selecionadas. É possível também que diversas combinações possam surgir nessa história, como: "Nome + Cidade Nascimento" ou "Idade e Cidade Nascimento".

Ao utilizar JPQL com consulta dinâmica teremos um problema bem chato de criar a `String`. Veja o código a seguir:

```
String consulta = "select c from Cachorro c where " +
if(nome != null){
    consulta += " c.nome = :nome"
}
```

```
if(idade != null){
    consulta += " c.idade = :idade"
}
// outros ifs
TypedQuery<Cachorro> query =
                entityManager.createQuery(consulta, Cachorro.class);

if(nome != null){
    query.setParameter("nome", nome);
}

if(idade != null){
    query.setParameter("idade",idade);
}

// outros códigos
```

Note no código como é feito primeiramente a montagem da consulta. É necessário verificar cada parâmetro enviado e adicionar o parâmetro na consulta. E por último, deve-se passar para cada um seu valor, novamente utilizando mais `ifs`.

O problema dessa abordagem é que é muito fácil ter um erro de sintaxe ao criar a *Query*. Veja a seguinte linha: `consulta += "c.idade = :idade "`. Note que o `c.idade` está grudado nas aspas e isso causaria erro de sintaxe, pois geraria algo como `where c.idade ....` Haverá uma perda de tempo precioso durante o tempo de desenvolvimento.

Outro problema é que se a query tiver os dois parâmetros, algo como a idade e o nome mostrado, pode acontecer de faltar colocar o termo `and` na consulta. A *query* acabaria ficando algo como `where c.idade = :idade c.nome = :nome`, faltando a condição `and`. Uma solução para esse problema seria deixar a query como o seguinte:

```
String consulta = "select c from Cachorro c where 1=1 " +
if(nome != null){
    consulta += " and c.nome = :nome"
}
if(idade != null){
    consulta += " and c.idade = :idade"
}
// outros ifs
TypedQuery<Cachorro> query =
```

```
                 entityManager.createQuery(consulta, Cachorro.class);

if(nome != null){
    query.setParameter("nome", nome);
}

if(idade != null){
    query.setParameter("idade",idade);
}

// outros códigos
```

Agora o nosso código permite que parâmetros sejam adicionados ou não na consulta.

Para esse tipo de situação em que uma consulta pode ter tantas possibilidades de erro na sintaxe, existe a chamada **Criteria**. É uma ótima solução, mas infelizmente de uso bem complexo.

A ideia da Criteria é justamente facilitar consultas dinâmicas otimizando a passagem de parâmetros com seus respectivos valores. Vamos começar com a rotina de listar tudo utilizando Criteria:

```
EntityManager entityManager = emf.createEntityManager();

CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();
CriteriaQuery<Cachorro> criteriaQuery =
                            criteriaBuilder.createQuery(Cachorro.class);
Root<Cachorro> root = criteriaQuery.from(Cachorro.class);
TypedQuery<Cachorro> query = entityManager.createQuery(criteriaQuery);
```

Esse código fará justamente um "select c from Cachorro c". O `CriteriaBuilder` é utilizado para criar o `CriteriaQuery` juntamente suas condições (`equals`, `greaterThan` etc.). A `CriteriaQuery` é utilizada para definir as condições relacionadas a um nível mais alto como: `order by`, `having`, `distinct`. O `Root` é utilizado como o caminho a ser percorrido pela consulta, ou seja, será sempre uma entidade que definirá quais os atributos relacionados a uma consulta. No exemplo, o `Root` foi definido como `Cachorro.class`, desse modo teríamos acesso a todos os relacionamentos a começar da entity `Cachorro`.

Mas ao olhar o código, um desenvolvedor pode perguntar: qual a vantagem de se utilizar esse tipo de código para uma consulta dinâmica?

Veja agora como realizar essas consultas com a Criteria:

```
EntityManager entityManager = emf.createEntityManager();

CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();
CriteriaQuery<Cachorro> criteriaQuery =
        criteriaBuilder.createQuery(Cachorro.class);
Root<Cachorro> root = criteriaQuery.from(Cachorro.class);

List<Predicate> condicoes = new ArrayList<Predicate>();

if(nome != null){
    Path<String> atributoNome = root.get("name");
    Predicate whereNome = criteriaBuilder.equal(atributoNome, nome);
    condicoes.add(whereNome);
}

if(idade != null){
    Path<Integer> atributoIdade = root.get("idade");
    Predicate whereNome = criteriaBuilder.equal(atributoIdade, idade);
    condicoes.add(whereNome);
}

Predicate[] condicoesComoArray =
                  condicoes.toArray(new Predicate[condicoes.size()]);

Predicate todasCondicoes = criteriaBuilder.and(condicoesComoArray);

criteriaQuery.where(todasCondicoes);

TypedQuery<Cachorro> query = entityManager.createQuery(criteriaQuery);
```

Apesar do código extremamente extenso, é possível ver agora como a consulta dinâmica começa a facilitar a vida. Não foi necessário adicionar o parâmetro e depois o valor. Automaticamente a própria JPA fez essa tarefa.

É justamente nesse enorme código acima que foi possível ver a desvantagem da Criteria da JPA. É muito código e muita complexidade para realizar apenas uma consulta com parâmetros dinâmicos.

Outro problema é que, do modo como foi feito, caso o atributo `idade` mudasse de nome para `age` todos os códigos que utilizassem `idade` deveriam mudar. Alterar

manualmente atributos em *Criteria* é bastante arriscado e sujeito a erros. Existe a alternativa de gerar classes com o *MetaModel* das entidades.

Um *MetaModel* de uma entidade nada mais é do que uma representação dos atributos dela, capaz de informar em tempo de compilação se existe algum erro. Ao realizar a alteração de `idade` para `age` as consultas que utilizassem o valor errado seriam acusadas em tempo de compilação.

Veja a classe com o metamodel de `Cachorro`:

```
@Generated(
    value ="org.hibernate.jpamodelgen.JPAMetaModelEntityProcessor"
)
@StaticMetamodel(Cachorro.class)
public abstract class Cachorro_ {
    public static volatile SingularAttribute<Cachorro, Integer> id;
    public static volatile SingularAttribute<Cachorro, String> name;
    public static volatile SingularAttribute<Cachorro, Integer> idade;
    // outras coisas
}
```

Para utilizar o *MetaModel*, uma classe deve ser gerada contendo todas as informações relativas à entity. Desse jeito, é possível reescrever a consulta exibida aqui do seguinte modo:

```
if(nome != null){
    Path<String> atributoNome = root.get(Cachorro_.name);
    Predicate whereNome = criteriaBuilder.equal(atributoNome, nome);
    condicoes.add(whereNome);
}

if(idade != null){
    Path<Integer> atributoIdade = root.get(Cachorro_.idade);
    Predicate whereNome = criteriaBuilder.equal(atributoIdade, idade);
    condicoes.add(whereNome);
}
```

Note que a mudança foi feita na linha `root.get(Cachorro_.idade)`. A partir de agora qualquer alteração será detectada e não haverá perda de tempo no desenvolvimento ou então problemas que estourariam na frente do usuário apontando que o atributo não existe.

Cada implementação da JPA costuma ter seu gerador de *MetaModel*:

- EclipseLink - CanonicalModelProcessor
- Hibernate - JPAMetaModelEntityProcessor
- OpenJPA - AnnotationProcessor6

Para utilizar com *Maven* e *Hibernate*, adicione a seguinte dependência ao *pom.xml*:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-jpamodelgen</artifactId>
    <version>1.2.0.Final</version>
</dependency>
```

E para que o *Maven* possa gerar essas classes, automaticamente adicione o plugin:

```
<plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
        <source>1.6</source>
        <target>1.6</target>
        <compilerArguments>
            <processor>
                org.hibernate.jpamodelgen.JPAMetaModelEntityProcessor
            </processor>
        </compilerArguments>
    </configuration>
</plugin>
```

Com o *plugin*, a classe de *MetaModel* seria automaticamente gerada.

Infelizmente, a Criteria nativa da JPA é complexa e verbosa, de difícil manutenção e o código entre as implementações pode variar. Para solucionar essa quantidade desvantagens existem frameworks especializados na Criteria da JPA, e desse modo a utilização ficaria mais simples.

A API de Criteria da JPA é extremamente complexa e trabalhosa. Para descrevê-la adequadamente, seria necessário um novo livro, dedicado apenas a ela.

7.1 A Criteria mais simples do Hibernate

A primeira solução é a Criteria do Hibernate. Veja a seguir como ficaria mais simples a mesma consulta vista anteriormente:

```
Criteria criteria = session.createCriteria(Cachorro.class);

if(name != null){
    criteria.add(Restrictions.eq("nome", nome));
}

if(idade != null){
    criteria.add(Restrictions.eq("idade", idade));
}

List<Cachorro> list = criteria.list();
```

O código está bem mais simples e fácil de utilizar. A Criteria do Hibernate é uma excelente solução para consultas dinâmicas.

O único *problema* é que seu código vai ficar preso ao *Hibernate*. Para utilizar o código, o seguinte `import` teria que ser utilizado:

```
import org.hibernate.Criteria;
import org.hibernate.Session;
import org.hibernate.criterion.Restrictions;
```

Na maioria das vezes, isso não seria problema uma vez que trocar implementação não é algo comum.

Outra ótima funcionalidade do *Hibernate* seria a chamada *Example Query*. Com esse tipo de Criteria você cria o objeto que quer pesquisar e depois dispara a pesquisa:

```
Pessoa pessoa = new Pessoa();
pessoa.setSexo('M');
pessoa.setIdade(33);
List results = session.createCriteria(Pessoa.class)
    .add( Example.create(pessoa) )
    .list();
```

Veja que foi instanciado um objeto do tipo `Pessoa` como os parâmetros desejados e depois disparada a pesquisa. É uma funcionalidade muito boa e prática.

7.2 EasyCriteria

Devido a esse problema de *Criteria* eu optei por criar um framework que trabalhasse com a *Criteria* da JPA de modo simples; a proposta desse framework é de facilitar a

utilização da Criteria da JPA e ainda assim manter a portabilidade do projeto entre suas maiores implementações: *OpenJPA*, *EclipseLink* e *Hibernate*.

A consulta que foi feita aqui com Criteria da JPA Nativa e *Hibernate* seria feita como a seguir utilizando o *EasyCriteria*:

```
EntityManager em = getEntityManagerFactory().createEntityManager();

EasyCriteria easyCriteria =
          EasyCriteriaFactory.createQueryCriteria(em, Cachorro.class);

if(nome != null){
    easyCriteria.andEquals("nome", nome);
}

if(idade != null){
    easyCriteria.andEquals("idade", idade);
}

List<Cachorro> resultList = easyCriteria.getResultList();
```

Veja que o *EasyCriteria* é bem simples e fácil de utilizar. Com ele é possível paginar consultas, contar o total de registros no banco de dados e ele também apresenta o interessante conceito de *CTO* (*Criteria Transfer Object*).

O *CTO* tem a ideia de facilitar o envio de parâmetros da camada de *View* para o local onde é realizada a consulta:

```
EasyCriteria easyCTO = EasyCriteriaFactory.createEasyCTO();

if(nome != null){
    easyCTO.andEquals("name", nome);
}

if(idade != null){
    easyCTO.andEquals("id", idade);
}
```

Note no código que a consulta pode ser criada em qualquer lugar, pois não necessita de um `EntityManager`. E para executar a consulta bastaria fazer:

```
EasyCriteria easyCriteria =
   EasyCriteriaFactory.createQueryCriteria(em, Cachorro.class, easyCTO);
```

```
List<Cachorro> resultList = easyCriteria.getResultList();
```

Um detalhe interessante do *EasyCriteria* é que ele tem 100% de cobertura em testes de unidade e pode ser baixado diretamente do *Maven* com a dependência que segue:

```
<dependency>
    <groupId>uaihebert.com</groupId>
    <artifactId>EasyCriteria</artifactId>
    <version>3.0.0</version>
</dependency>
```

O framework ainda é novo e tem muito que crescer, mas para a maioria das consultas utilizadas no dia a dia ele já consegue atender satisfatoriamente.

Seu site oficial é: http://easycriteria.uaihebert.com

Capítulo 8

Recursos avançados com a JPA

8.1 Não deixe os resultados da consulta em memória

A JPA pode ter uma performance otimizada ao realizar consultas cujos resultados servirão apenas para leitura. Imagine que o sistema terá 30 relatórios que apenas exibirão praticamente os mesmos dados para o usuário. Nesse caso, é possível usar configuração de transação, que otimizará a consulta.

Vamos ver como otimizar com as consultas quando se utiliza EJB, Spring e um programa em ambiente Java SE. Os conceitos são os mesmos, independente da tecnologia adotada.

8.1.1 Otimização com EJB

Veja o código a seguir executado por um EJB:

```
@Stateless
public class PessoaDAO {
```

```
    @PersistenceContext(unitName = "myPU")
    private EntityManager entityManager;

    public void editarNome(Integer id, String novoNome){
        Pessoa pessoa = entityManager.getReference(Pessoa.class, id);
        pessoa.setName(novoNome);
    }

    public List<Pessoa> listarTodos(){
        TypedQuery<Pessoa> query = entityManager.
                    createQuery("select p from Pessoa p", Pessoa.class);

        return query.getResultList();
    }

    public List<Pessoa> listarTodosSemCachorro(){
        String consulta = "select p from Pessoa p " +
                          "where p.cachorros is empty";

        TypedQuery<Pessoa> query = entityManager.
                    createQuery(consulta, Pessoa.class);
        return query.getResultList();
    }
}
```

No código da classe `PessoaDAO`, é possível ver um EJB que está tomando conta da transação. O comportamento padrão da JPA é que toda entity que passe pelo `EntityManager` estará *managed*, e depois pode virar *detached*, além de poder transitar pelos outros estados.

No `PessoaDAO`, após retornar uma lista com o método `getResultList()`, todas as entidades estarão *managed* e serão armazenadas na memória.

Mas qual é o sentido de deixar as entidades como *managed* se elas serão apenas utilizadas em um relatório? Ou utilizadas em uma única tela?

Para otimizar as consultas, basta informar à JPA que aquele método não deve ter seu resultado *managed*, ou seja, todas as *Entities* retornadas já devem estar *detached*. Veja como ficará o código já otimizando os métodos que servem apenas para leitura:

```
@Stateless
public class PessoaDAO {
```

```
    // outras coisas

    public void editarNome(Integer id, String novoNome){
        Pessoa pessoa = entityManager.getReference(Pessoa.class, id);
        pessoa.setName(novoNome);
    }

    @TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
    public List<Pessoa> listarTodos(){
        TypedQuery<Pessoa> query = entityManager.
                    createQuery("select p from Pessoa p", Pessoa.class);

        return query.getResultList();
    }

    @TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
    public List<Pessoa> listarTodosSemCachorro(){
        String consulta = "select p from Pessoa p " +
                          "where p.cachorros is empty";

        TypedQuery<Pessoa> query = entityManager.
                    createQuery(consulta, Pessoa.class);
        return query.getResultList();
    }
}
```

Agora a anotação `@TransactionAttribute` está presente, informando que uma transação não será necessária para aquela consulta. Quando um método anotado com `TransactionAttributeType.NOT_SUPPORTED` é executado, toda entidade que passar pelo `EntityManager` já estará por default *detached*. Desse modo, podemos otimizar as consultas realizadas.

A mesma classe que vimos poderia também ser configurada como:

```
@Stateless
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public class PessoaDAO {
    // outras coisas

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void editarNome(Integer id, String novoNome){
        Pessoa pessoa = entityManager.getReference(Pessoa.class, id);
```

```
        pessoa.setName(novoNome);
    }

    public List<Pessoa> listarTodos(){
        TypedQuery<Pessoa> query = entityManager.
                   createQuery("select p from Pessoa p", Pessoa.class);

        return query.getResultList();
    }

    public List<Pessoa> listarTodosSemCachorro(){
        String consulta = "select p from Pessoa p " +
                          "where p.cachorros is empty";

        TypedQuery<Pessoa> query = entityManager.
                   createQuery(consulta, Pessoa.class);
        return query.getResultList();
    }
}
```

O EJB inteiro foi anotado com `TransactionAttributeType.NOT_SUPPORTED` e ele terá o comportamento de não ter transação. O método `editarNome` necessita de transação e por isso foi anotado com `TransactionAttributeType.REQUIRED` sobrescrevendo o comportamento do EJB.

8.1.2 Otimização com Spring

O Spring adota um comportamento semelhante ao do EJB, mas com anotações diferentes:

```
@Repository
@Transactional(readOnly = true)
public class PessoaDAO {
    // outras coisas

    @Transactional(readOnly = false)
    public void editarNome(Integer id, String novoNome){
        Pessoa pessoa = entityManager.getReference(Pessoa.class, id);
        pessoa.setName(novoNome);
    }
```

```
    public List<Pessoa> listarTodos(){
        TypedQuery<Pessoa> query = entityManager.
                    createQuery("select p from Pessoa p", Pessoa.class);

        return query.getResultList();
    }

    public List<Pessoa> listarTodosSemCachorro(){
        String consulta = "select p from Pessoa p " +
                          "where p.cachorros is empty";

        TypedQuery<Pessoa> query = entityManager.
                    createQuery(consulta, Pessoa.class);
        return query.getResultList();
    }
}
```

Veja que para o Spring também existe o conceito de transação e também como deixar a transação receber entidades já no estado *detached*.

8.1.3 Java SE ou transação manual

Quando temos a transação criada manualmente ou em ambiente Java SE, bastaria fazer como:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("PU");
EntityManager entityManager = emf.createEntityManager();

TypedQuery<Pessoa> query = entityManager.
   createQuery("select p from Pessoa p", Pessoa.class);

query.getResultList();
```

Veja que o `EntityManagerFactory` foi criado e logo depois um `EntityManager`. O detalhe é que a `query` foi executada sem abrir uma transação, o comando `entityManager.getTransaction().begin();` não foi executado.

Quando falamos de transação manual ou ambiente JSE (onde acabamos por controlar a criação do *EntityManagerFactory*), para consultas não é necessário abrir uma

transação. Ao disparar a consulta, a JPA verificará se existe alguma transação, e não encontrando, ele trabalhará com uma consulta apenas de leitura para trazer as informações.

8.2 Paginação de consultas

Uma prática muito comum em rotinas que precisam acessar muitos dados em uma tabela é a paginação.

Imagine uma tela em que o usuário queira visualizar todos os seus registros de SMS enviados e recebidos. Um adolescente com um pacote de mensagens generoso poderia mandar milhares de mensagens em uma semana. Ao recuperar o resultado de todas essas mensagens do banco de dados, a memória do servidor possivelmente estouraria.

Para resolver esse problema, existe a técnica de paginar o resultado de uma consulta, ou seja, pequenos pedaços dos resultados seriam trazidos para o usuário. Veja a imagem:

Total consumo celular: (33) 3333-3333

(1 of 100) 1 2 3 4 5 6 7 8 9 10 10

Id	Tipo	Total Letras
0	SMS	57
1	SMS	47
2	SMS	30
3	SMS	96
4	SMS	49
5	SMS	96
6	SMS	7
7	SMS	48
8	SMS	82
9	SMS	77

(1 of 100) 1 2 3 4 5 6 7 8 9 10 10

Figura 8.1: Registros de SMS de um usuário

A cada página nova solicitada pelo usuário, uma nova consulta seria feita no banco de dados.

A JPA já consegue tratar essa paginação nativamente, como pode ser visto no código a seguir:

```
String consulta = "select s from SMS s " +
                  "where s.celular.numero = '(33) 3333-3333'";

TypedQuery<SMS> query = entityManager.createQuery(consulta, SMS.class);

query.setMaxResults(5);
query.setFirstResult(0);
```

Note que a consulta tem a mesma sintaxe. O que mudou foi a chamada a dois métodos: `setMaxResults` e `setFirstResult`.

O `setMaxResults` determina quantos resultados voltarão ao executar o método `getResultList`. Já o `setFirstResult` define a partir de qual índice se deve começar a retornar as entidades. Ao definir `setFirstResult` como zero, os primeiros 5 resultados virão; se alterar para `query.setFirstResult(5)` a JPA trará apenas os resultados de `index > 4`, ou seja, pulará os primeiros resultados.

A paginação é a melhor solução quando temos muitos dados retornados e poucos são utilizados e visualizados.

8.3 Operações em muitos registros - Bulk Operations

Em diversos momentos é necessário realizar uma operação em muitos registros do banco de dados. Imagine uma base na qual todos os usuários cadastrados antes de 2012 devem ser desativados, ou então, todos os clientes cadastrados receberão um bônus de 30 reais diários.

Note que, quando falamos de operações em muitos registros, também conhecido como *bulk operations*, estamos falando de uma quantidade geralmente desconhecida de registros a serem afetados. É possível que a rotina atualize apenas 33 registros como também atualize mais de 333 mil. Como essa rotina é bem delicada, vamos ver com calma os devidos cuidados a se tomar.

Para executar um bulk update, é simples. Veja o código:

```
String bulkUpdate =
    "update Usuario s set s.enabled = false " +
```

```
    "where s.dataRegistro < '01/01/2012'";
Query query = em.createQuery(bulkUpdate);
query.executeUpdate();
```

Note que esse código realizará um update no banco de dados, independentemente da quantidade de registros existentes lá.

Até então uma ação simples, mas e no caso de uma exclusão?

```
String bulkUpdate =
    "delete from Usuario s " +
    "where s.tempoInativacao > 2000 ";
Query query = em.createQuery(bulkUpdate);
query.executeUpdate();
```

Note que um `Usuario` será excluído baseado em seu tempo de inativação. É agora que o primeiro problema pode aparecer. E se `Usuario` tiver um relacionamento com outra entity? Imagine um relacionamento `@ManyToMany` com uma entity chamada `Perfil`. Ou seja, no banco de dados existe um relacionamento de chaves onde uma exceção acontecerá caso o relacionamento não seja desfeito.

É nessa hora que vem o seguinte pensamento em mente: "uma solução simples seria então adicionar `CascadeType.REMOVE` no relacionamento que estaria tudo resolvido!!!". Aí vem a bomba: operações bulk update não disparam o evento *Cascade*. Qual a solução então? Primeiro executar um comando que elimine o relacionamento e só depois o `delete` na entity `Usuario`:

```
String bulkUpdate =
    "Update Usuario s " +
    "set s.perfis = null " +
    "where s.tempoInativacao > 2000";
Query query = em.createQuery(bulkUpdate);
query.executeUpdate();
```

Veja que no código anterior primeiramente é eliminado o relacionamento e só então depois que poderemos executar o `delete` sem problemas.

Cuidado

Ao utilizar *bulk operations* todo o controle de relacionamentos fica por conta do desenvolvedor. Tenha certeza de que todos os relacionamentos que dependem de chave estrangeira estão sendo corretamente tratados.

Outro cuidado que precisamos ter é com relação a duração de uma transação. Veja o código:

```
entityManager.getTransaction().begin();

// atualiza diversos usuários
String bulkUpdate =
    "update Usuario s set s.enabled = false ";
Query query = entityManager.createQuery(bulkUpdate);
query.executeUpdate();

// busca um usuário no banco de dados
Usuario usuario = entityManager.find(Usuario.class, 33);
System.out.println(usuario.isEnabled());

// finaliza a transação
entityManager.getTransaction().commit();
```

Aqui temos uma busca de um `Usuario` de `id = 33` sendo trazido do banco de dados. Note que, no comando, todos os usuários do banco de dados estão sendo desabilitados. O que aconteceria quando o `println` fosse invocado, uma vez que antes do *bulk update* todos os status eram `enable como true`? Note que um bulk update acontece e, logo após, o `Usuario` que foi recuperado do banco de dados teve seu *status* utilizado: `System.out.println(usuario.isEnabled()).`

Respondendo a pergunta: "o status do usuário continuaria como true". Por quê? Toda ação do tipo *bulk* não é gerenciada pelo *Persistence Context*, ou seja, o comando do `update` ainda não foi aplicado no banco de dados e com isso o *EntityManager* busca por dados desatualizados. Toda ação *bulk* será refletida diretamente no banco de dados quando o comando for disparado, e não será monitorado pelo *PersistenceContext*.

Imagine o seguinte cenário:

- transação inicia;
- usuário de id 33 recuperada do banco de dados;
- bulk delete de todos usuários são excluídos;
- usuário de id 33 recebe `status = enabled`;
- busca por todos usuários enabled;
- commit.

Note que serão realizados ação de consulta, `update`, `bulk delete` e busca novamente. E todas essas ações de bulk podem ocasionar em informações desatualizadas como visto anteriormente. E pior, essa ação pode gerar resultados diferentes. No Hibernate, por exemplo, houve erro na execução do batch. Já no EclipseLink não aconteceu erro algum e na OpenJPA houve erro de concorrência.

> **Cuidado**
>
> Toda vez que tratamos com *bulk operations* é necessário haver um controle cuidadoso da transação.

Podemos utilizar duas estratégias para evitar esse tipo de problema. A primeira solução é sempre executar essa operação em uma transação separada, ou seja, abrir a transação executar a operação em massa e finalizá-la. A outra opção é: antes de cada operação em massa, chamar o comando `entityManager.clear()` para forçar o esvaziamento dos objetos que estão alocados no *PersistenceContext*. Após esse esvaziamento, nenhum objeto estará mais em memória.

> **Cuidado**
>
> Tenha muito cuidado ao chamar o comando `entityManager.clear()`, pois ele fará com que não haja nenhuma entity no *PersistenceContext*, o que obrigará o `entityManager` a fazer uma nova consulta caso alguma entity seja solicitada.

A `bulk operation` existe e pode facilitar em muita coisa a vida do desenvolvedor, mas deve sempre ser utilizada com muita cautela.

8.4 Tratamento de concorrência

Imagine uma clínica veterinária que atende 150 cachorros por hora. Aí uma linda vira-lata dá entrada para ser atendida onde já havia sido tratada antes. E uma ficha cadastral como a seguinte aparece na tela do funcionário:

Figura 8.2: Ficha de Cadastro

Suponha que a atendente "A" abre essa tela e prepara a edição dos dados da paciente `Minhoca`. Contudo, por um pequeno descuido da secretária, a atendente "B" também fica encarregada de editar os dados do atendimento e atualização cadastral. Só que ambas começam as edições com poucos segundos de diferença, mas a atendente "A" altera apenas os dados cadastrais e se prepara para apertar o botão salvar. Enquanto isso, a atendente "B" atualiza apenas os dados dos atendimentos e o telefone do dono da paciente e também se prepara para apertar o botão salvar.

O que acontecerá quando, por obra da "Lei de Murphy", ambas atendentes clicarem no mesmo botão às 13:33:33 333s? É possível dizer que: alguém perderia "dados" nessa história.

Esse é um problema comum quando falamos de muitas pessoas acessando a mesma informação em determinado momento. Quanto maior o número de pessoas

acessando o mesmo sistema, maior a chance de isso acontecer.

Existem alguns problemas que podem acontecer quando existem muitos acessos simultâneos ao banco de dados e no mesmo registro. Os erros que podem acontecer quando falamos de acessos simultâneos são:

- *Dirty Read* — é um dos problemas mais graves que podem acontecer ao acessar o banco de dados. Imagine que uma transação, ao ler informações no banco de dados, acabe lendo as que ainda não receberam o *commit* de outra transação. A atendente "A" busca um relatório que trará dados de um cadastro que ainda não foi finalizado pela atendente "B", ou seja, a transação da atendente "A" lerá dados que ainda não receberam *commit* da transação da atendente "B". Caso a transação da atendente "B" dê *rollback* esses dados vão se perder;
- *Repeated Read/Unrepeated Read* — acontece justamente quando a mesma transação faz a mesma consulta ao banco de dados duas ou mais vezes mas os resultados retornados são diferentes.

A JPA apresenta o conceito de *Lock* segundo o qual é possível que uma transação avise a toda aplicação algo como: "esse registro do banco de dados é meu, ninguém mais deve alterar". Realizar um *Lock* nada mais é que travar determinado registro até que ele seja liberado.

Existem dois tipos de *Lock*: o "Pessimista" e "Otimista". A diferença entre eles é bem simples.

O *Lock* otimista acredita que aquele registro não será utilizado enquanto sua transação estiver ativa, então, o registro será travado apenas ao final da transação. Essa é a abordagem mais utilizada por ser a que mais beneficia o desempenho da aplicação.

Já o *Lock* pessimista acredita que, a qualquer momento durante a transação, seus objetos podem ser alterados, e desse modo, ele já trava o objeto durante toda a transação. Ele é normalmente utilizado em operações críticas nas quais esse registro tem que se manter íntegro durante toda a transação.

Enquanto um dado está travado, ele não poderá ser editado ou até mesmo lido por outra *transação* se não pela que que executou a trava. A abordagem de *Lock* é muito útil, mas deve ser utilizada com cuidado.

Veremos mais à frente os tipos de *Lock* para ambas abordagens. Para já ter algum tipo de segurança com persistência simultânea é possível utilizar o recurso chamado de *Version*, que nada mais é que um campo autoincremento comandado pela JPA

para saber qual a versão atual da entity. Veja abaixo como ficaria o *Version* em uma entity:

```
@Entity
public class Cachorro {
    @Id
    private int id;
    private String nome;

    @Version
    private int version;
    // outras coisas
}
```

Note que para definir um campo como *Version* é bastante simples e no caso da concorrência é muito útil. Esse campo funciona do seguinte modo:

1) Suponhamos que a *Minhoca* está com `version = 1`;

2) O telefone de contato da *Minhoca* muda;

3) Ao realizar o persist da *Minhoca* o campo version iria para 2.

Como o campo `version` ajudaria nesse caso? Vamos rever o problema acima citado no começo do capítulo:

1) Atendente "A" busca a *Entity Minhoca* com `version = 1`;

2) Atendente "B" busca a *Entity Minhoca* com `version = 1`;

3) Atendente "A" consegue realizar corretamente o *commit* o que faz com que o version da *Minhoca* vá para 2;

4) Atendente "B" vai ter uma mensagem de erro, pois a versão do objeto que ela tem é de valor 1, mas no banco já consta que a versão atual é 2.

Dica

Algumas implementações não exigem o uso `@Version` para o controle de versão ao utilizar *Lock* otimista. O ideal é ter em suas classes o `@Version` para manter a portabilidade entre as implementações quando necessário.

Um detalhe sobre o `@Version` é que uma alteração em relacionamento da entity pode não ser refletida no banco de dados. Imagine o seguinte: cada atendente "A" e "B" está com a entidade *Minhoca* com `version = 33` em memória. Acontece que a atendente "A" adiciona um brinquedo na lista de brinquedos da *Minhoca* (relacionamento `@ManyToMany`), e é aí que pode começar o problema.

Quando a alteração não reflete em alguma coluna da entidade, o campo version não será alterado. Note que a alteração que aconteceu foi um `insert` em uma tabela de relacionamento e nada mais (`CACHORRO_TEM_BRINQUEDO` composta de `cachoro_id` e `brinquedo_id`). Desse modo, a JPA não marca que a entity teve seu *@Version* alterado.

É preciso ter bastante cuidado quando falamos de *Bulk Operation* e `@Version`. Assim como uma *Bulk Operation* não inicia o Cascade ele também não incrementa o `@Version`.

Para atualizar um registro com o comando `update` e juntamente seu `@Version` faça como o que segue:

```
UPDATE Usuario u
SET u.version = u.version + 1, u.enabled = false
WHERE id = 33
```

8.4.1 Read Committed

Esse é o padrão da JPA já para evitar o problema do *Dirty Read* que vimos acima. A JPA já utilizará a abordagem chamada *Read Committed* e não lerá alterações que não receberam commit ainda.

8.4.2 Aplicando o Lock

Existe a opção de bloquear a informação a nível de leitura. Desse modo, seria possível buscar informações sem correr o risco de acontecer o problema de *Repeated Read*.

É possível aplicar *Lock* de quatro modos na JPA: `EntityManager.find(...)`, `EntityManager.refresh(...)`, `EntityManager.lock(...)` e `Query.setLockMode(...)`.

O `EntityManager.find(...)` indica à JPA para marcar com *Lock* já na hora que a informação for trazida do banco de dados. Para aplicar o *Lock*, poderia ser feito algo como:

```
LockModeType lockType = LockModeType.OPTIMISTIC;
Usuario usuario = entityManager.find(Usuario.class, 33, lockType);
```

A partir do momento em que o usuário de `id = 33` for recuperado do banco de dados, ele está marcado com *Lock*.

`EntityManager.refresh(...)` fará com que a JPA sincronize os dados do usuário com o banco de dados e indicará que esse objeto será marcado com *Lock*:

```
LockModeType lockType = LockModeType.OPTIMISTIC;
Usuario usuario = entityManager.refresh(Usuario.class, 33, lockType);
```

`EntityManager.lock(...)` define uma entity presente no *Persistence Context* do `EntityManager` será marcada para *Lock*:

```
LockModeType lockType = LockModeType.OPTIMISTIC;
entityManager.refresh(usuario, lockType);
```

`Query.setLockMode(...)` fará com que todo resultado retornado da consulta seja marcado para *Lock*:

```
LockModeType lockType = LockModeType.OPTIMISTIC;
TypedQuery<Usuario> query =
   entityManager.createQuery("select u from Usuario u", Usuario.class);
query.setLockMode(lockType);
```

Note que foi dito que os objetos seriam marcados para *Lock* e não receberiam o *Lock* já de cara. O comportamento do *Lock* varia da abordagem pessimista e otimista que veremos a seguir.

8.4.3 Lock Otimista

O *Lock* otimista pode acontecer em qualquer momento da transação, mas será aplicado apenas no momento do commit. Veja o código:

```
// inicia a transação
LockModeType lockType = LockModeType.OPTIMISTIC;
Usuario usuario = entityManager.find(Usuario.class, 33, lockType);

// realiza toda a tarefa

// commit será realizado
```

No código foi possível ver que logo após iniciar a transação o *Lock* `OPTIMISTIC` foi realizado, mas ele só terá efeito na hora do *commit*. Não faz diferença a hora em

que esse comando for executado, ele só terá efeito quando o commit for efetivado. É justamente por isso que há diferença entre usar otimista e pessimista. A performance é muito boa, pois o *Lock* só é tratado ao final da transação, juntamente com o *commit*.

Existe também uma forma de forçar a alteração do `@Version` da entity ao fazer o *Lock*:

```
LockModeType lockType = LockModeType.OPTIMISTIC_FORCE_INCREMENT;
Usuario usuario = entityManager.find(Usuario.class, 33, lockType);
```

Desse modo, ao realizar o *Lock*, o *@Version* será automaticamente atualizado.

8.4.4 Lock Pessimista

Esse *Lock* tem efeito a partir do momento em que ele foi utilizado. Veja o código:

```
// inicia a transação
LockModeType lockType = LockModeType.PESSIMISTIC_READ;
Usuario usuario = entityManager.find(Usuario.class, 33, lockType);

// faz mil coisas

// commit será realizado
```

No código visto, a partir do momento em que o *Lock* foi informado, o registro referente a esse *Lock* é travado até o final da transação.

Pode-se também utilizar o *Lock* para escrita `LockModeType.Write` ou para travar o registro e atualizar seu `@Version` com `LockModeType.PESSIMISTIC_FORCE_INCREMENT`.

É possível também definir Timeouts para evitar de uma transação ficar muito tempo em espera:

```
Map<String,Object> opcoes = new HashMap<String,Object>();
opcoes.put("javax.persistence.lock.timeout", 5000);
em.find(Usuario.class, 33, LockModeType.PESSIMISTIC_WRITE, opcoes);
```

Considerações Finais

Infelizmente não existe uma forma certa e definida de como cada *Lock* trabalhará, isso varia de acordo com a implementação e situação. Por exemplo, quando um registro estiver definido com *Lock* e uma outra transação tentar acesso a ele, uma

Exception pode acontecer, ou então a transação poderá entrar em espera e, não conseguindo acesso, pode acontecer a *Exception*.

Lock pessimista tem que ser utilizado com muito, mas muito cuidado mesmo e somente quando necessário dentro da transação. Imagine o cenário:

1) transação inicia;

2) consulta para gerar relatorio 1 (leva 5 minutos);

3) consulta para gerar relatorio 2 (leva 7 minutos);

4) processamento que atualizará os registros;

5) fim da transação.

Nesse cenário, note que apenas um processamento realiza alteração dos dados, o *Lock* pessimista poderia ser aplicado apenas nesse exato momento para evitar que demais dados sejam protegidos de acesso e alteração.

Capítulo 9

Finalizando

A JPA tem diversas funcionalidades que podem ou não facilitar a vida do desenvolvedor. É necessário estar atento ao seu funcionamento para utilizá-la da maneira correta. Diversos problemas podem acontecer quando a JPA é utilizada de modo incorreto, o que leva pessoas a criticarem a ferramenta, sendo que o problema está em quem a utiliza.

JPA não é a bala de prata quando falamos de ORM, mas sim uma mão na roda que ajuda no desenvolvimento.

Espero que vocês tenham gostado do livro, até mais! \o_